Desarrollo de aplicaciones multiplataforma

MÓDULO:

Programación

UNIDAD:

Utilización de objetos

CEAC
FP Oficial

MÓDULO:
Programación

UNIDAD:
Utilización de objetos

CEAC FP Oficial

ÍNDICE

1. Introducción

A lo largo de esta unidad estudiaremos las bases de la programación orien-tada a **objetos** (POO o, en sus siglas en inglés, OOP, ***Object-oriented pro-gramming***).

La programación orientada a objetos debe tomarse como una filosofía, un modelo de programación. A continuación, estudiaremos los conceptos bá-sicos que se aplican a la POO desde un punto de vista general, sin particu-larizar en ningún lenguaje de programación específico.

Es muy importante señalar que cuando hacemos referencia a la programa-ción orientada a objetos no estamos hablando de características añadidas a un lenguaje de programación, sino de una nueva forma de pensar acerca del proceso de descomposición del problema y de desarrollo de soluciones.

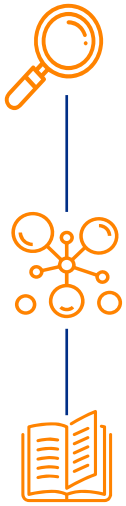
La POO propone una serie de recursos basados en entidades del mundo real para la construcción de estructuras y soluciones. La POO surge como un intento para controlar la complejidad inherente al desarrollo de aplica-ciones informáticas.

En POO, un objeto es la abstracción de un elemento real, al que se dota de atributos representados por sus características o propiedades, y que representan las acciones que realiza. Si aprendemos el uso de los objetos seremos capaces de crear códigos más simplificados, reutilizables y que propiciarán el mejor funcionamiento de nuestro programa.

El conocimiento de los conceptos y procedimientos que se desarrollan en esta unidad, es una buena base de partida para el aprendizaje de cualquier lenguaje de programación orientado a objetos. En cada uno de ellos, las instrucciones pueden ser diferentes, pero aunque cambie el lenguaje, los conceptos a manejar serán prácticamente los mismos.

2. Objetivos pedagógicos

Los objetivos de aprendizaje que se pretenden alcanzar con esta unidad son:



Comprender los Fundamentos de la Programación Orientada a Objetos.

Utilizar Recursos Basados en Entidades del Mundo Real.

Establecer una Base para el Aprendizaje de Lenguajes de Programación Orientados a Objetos.

3. Utilización de objetos

3. 1. Características de los objetos

En programación, un **objeto** es una entidad provista de un conjunto de **propiedades** (datos) y de **comportamientos** (métodos) que reaccionan según determinados eventos. Un objeto puede ser, tanto la representación informática de un concepto del mundo real, como la de objetos internos de un programa informático.

Los objetos tienen tres características principales:

- **Estado**: El estado de un objeto se refiere al conjunto de datos que pertenecen al objeto, es decir, sus propiedades o variables. Estas propiedades pueden almacenar valores específicos que describen las características del objeto en un momento dado. El estado de un objeto puede cambiar a lo largo de la ejecución del programa, ya sea mediante la asignación directa de valores a las propiedades o a través de la ejecución de métodos que modifican esas propiedades.
- **Comportamiento**: El comportamiento de un objeto está definido por el conjunto de métodos que contiene. Los métodos representan las operaciones o acciones que el objeto puede realizar. Pueden manipular el estado del objeto, interactuar con otros objetos, realizar cálculos, procesar datos, etc. Los métodos definen cómo se comporta y cómo responde el objeto frente a diferentes acciones o mensajes.
- **Identidad o nombre**: Cada objeto tiene una identidad única que lo distingue de otros objetos. La identidad se refiere a la característica que define al objeto como una entidad única dentro del programa informático. Puede ser utilizado para referirse al objeto a lo largo del código fuente y para distinguirlo de otros objetos similares. La identidad de un objeto se mantiene constante durante su ciclo de vida y no cambia, incluso si su estado o comportamiento se modifican.

Podemos considerar a los objetos entidades independientes dentro del código, que utilizaremos para realizar acciones que intervengan en el curso de nuestro programa.

Por ejemplo, podemos crear el objeto Impresora (esta será su identidad), que admitirá métodos como imprimir un documento, abortar una impresión, vaciar la cola de documentos a imprimir, o suprimir un documento de la cola de impresión. Su estado dependerá de si está imprimiendo y de los trabajos que tenga en la cola.

Coche	
Propiedades	Métodos
• **Color**	• Arrancar
• **Marca**	• Acelerar
• **Modelo**	• Frenar
• **Cilindrada**	• ...
• **Carburante**	
• **Velocidad...**	

Estructura básica (clase) de un objeto Coche.de un objeto Coche. e tabla donde guardamos tipos de alimentos por fabricantes.

En la programación orientada a objetos, los objetos pueden interactuar entre sí a través de relaciones. Estas relaciones pueden ser de varios tipos, como asociación, composición o herencia. Las relaciones entre objetos permiten modelar situaciones más complejas y representar la interacción entre diferentes entidades del sistema.

- **Asociación**: Los objetos se asocian cuando uno de ellos utiliza o tiene referencia al otro. Pueden ser asociaciones unidireccionales o bidireccionales.
- **Composición**: Es una relación fuerte en la que un objeto está compuesto por uno o más objetos más pequeños. Los objetos compuestos no pueden existir independientemente del objeto que los contiene.
- **Herencia**: Permite la creación de nuevas clases basadas en clases existentes. Una clase hereda propiedades y métodos de otra clase (clase padre o superclase), lo que permite reutilizar y extender el código.
- **Agregación**: Es similar a la composición, pero la relación es más débil. Los objetos agregados pueden existir de manera independiente al objeto que los contiene.
- **Dependencia**: Un objeto depende de otro cuando necesita utilizarlo en algún momento, pero no hay una relación de propiedad o asociación fuerte entre ellos.

3.2. Instanciación de objetos

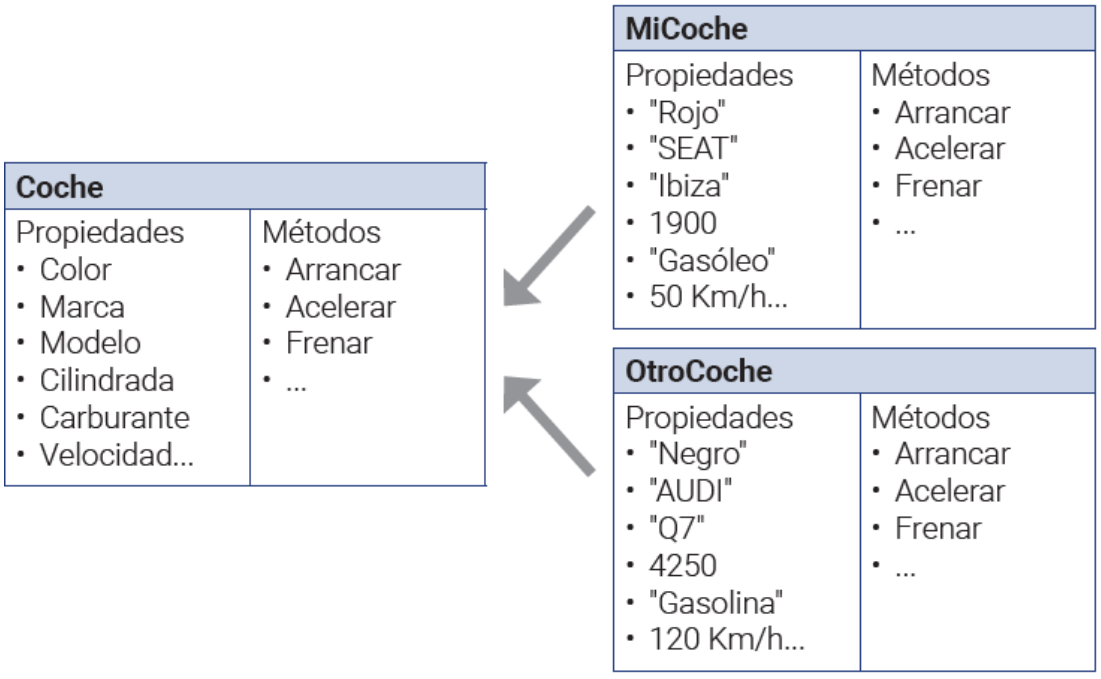
Los objetos se definen generando una instancia de una clase en la memoria. Una clase describe exactamente qué datos y métodos tendrá un determinado tipo de objeto.

La **instanciación** es el proceso mediante el cual se crea un objeto en el código y se obtiene una instancia única de una clase. Una vez que se ha instanciado un objeto, se pueden utilizar sus funciones (métodos) para realizar acciones y operaciones que intervengan en la ejecución del programa.

La instanciación implica dos pasos principales:

- **Creación del objeto:** Se utiliza el operador "new" seguido del nombre de la clase y, si es necesario, los argumentos para el constructor de la clase. Este proceso reserva memoria para el objeto y ejecuta el constructor de la clase para inicializar sus propiedades y establecer su estado inicial.
- **Obtención de la instancia:** El resultado de la creación del objeto es una instancia única de la clase, que se asigna a una variable o se utiliza directamente en el código. La variable o referencia a la instancia del objeto se utiliza para acceder a sus métodos y propiedades, permitiendo su uso en la ejecución del programa.

A través de la instancia del objeto, se pueden invocar los métodos y utilizar las funciones definidas en la clase. Esto permite que el objeto intervenga en la ejecución del programa, realizando tareas específicas, procesando datos, interactuando con otros objetos, etc.



Ejemplo de instancias (objetos) de una clase

A continuación se expone cómo se instanciarían estos objetos en diferen-tes lenguajes, sin asignar los valores y asignándolos:

- Sin asignación de las propiedades básicas:
  - Java: Coche miCoche = new Coche();
  - C++: Coche miCoche; o

    Coche *otroCoche = new Coche0;

- Asignando las propiedades básicas en el momento de creación:

- Java: Coche miCoche = new Coche("Rojo", "SEAT","Ibiza"…);
  - C++:
  - Coche miCoche("Rojo", "SEAT","Ibiza",….); o

    Coche *otroCoche = new Coche("Negro","Audi","Q7"…);

3.3. Utilización de métodos

La **invocación** de métodos de objetos es un aspecto fundamental en la programación orientada a objetos. Los métodos representan el comportamiento de un objeto y son responsables de realizar acciones o cálculos específicos en función de los datos del objeto.

Cuando se invoca un método de un objeto, se está llamando a su implementación específica para ese objeto en particular. La invocación de un método implica proporcionar los argumentos necesarios (si los hay) y realizar la llamada al método utilizando la sintaxis adecuada.

La sintaxis general para invocar un método de objeto es la siguiente:

```
objeto.metodo(argumentos);
```

En este ejemplo en C++, primero instanciaremos la clase Aritmética:

```
Aritmética calculadora;
```

A continuación podremos usar sus métodos:

```
resultadoSuma = calculadora.suma(96, 35);

resultadoMultiplicacion = calculadora.multiplicar(96, 35);

resultadoDivision = calculadora.dividir(96, 35);
```

calculadora hace referencia a la identidad, el nombre, del objeto que estamos instanciando. La instrucción definida tras la identidad es el método que vamos a usar, y los valores escritos entre paréntesis son los parámetros que usará el método para calcular el resultado. El valor obtenido en cada operación será almacenado en el dato correspondientemente definido.

```
class Aritmetica {
    public:
        inline int sumar (int a, int b) const
        {
            return a + b;
        }
        inline int restar (int a, int b) const
        {
            return a - b;
        }
        inline float multiplicar (int a, int b) const
        {
            return a * b
        }
}
```

Estructura de un objeto definido en lenguaje de programación C++ que almacena una colección de métodos para realizar

3.4. Utilización de propiedades

Las propiedades representan los datos o atributos que pertenecen a un objeto y se utilizan para almacenar información relevante sobre el estado del objeto.

Para acceder a las propiedades de un objeto, se utiliza la sintaxis de acceso al miembro, que consiste en el nombre del objeto seguido del operador de acceso ".", seguido del nombre de la propiedad que se desea acceder.

La sintaxis general para acceder a una propiedad de un objeto es la siguiente:

objeto.propiedad

3.4.1. Modificación de propiedades

La modificación de propiedades de objetos permite cambiar los valores de las variables internas de un objeto, lo que afecta su estado y comportamiento.

Para modificar una propiedad de un objeto, se utiliza la sintaxis de acceso al miembro, que consiste en el nombre del objeto seguido del operador de acceso ".", seguido del nombre de la propiedad que se desea modificar, y finalmente el operador de asignación "=" seguido del nuevo valor que se desea asignar a la propiedad.

La sintaxis general para modificar una propiedad de un objeto es la siguiente:

objeto.propiedad = nuevoValor;

Siguiendo el ejemplo del apartado anterior, en cuanto a las propiedades, se accede con la notación objeto.propiedad. Por ejemplo, la clase Coche podríamos forzar el valor de su propiedad "velocidad" mediante la siguiente sentencia:

miCoche.velocidad=50;

En caso de C++, si se hubiera creado mediante un puntero, se haría de la siguiente forma:

otroCoche->velocidad=100.

La forma objeto.propiedad es la usada siempre por Java.

También en C++ se accedería de una de las siguientes formas, según fuera vía un objeto o un puntero:

miCoche.acelerar();

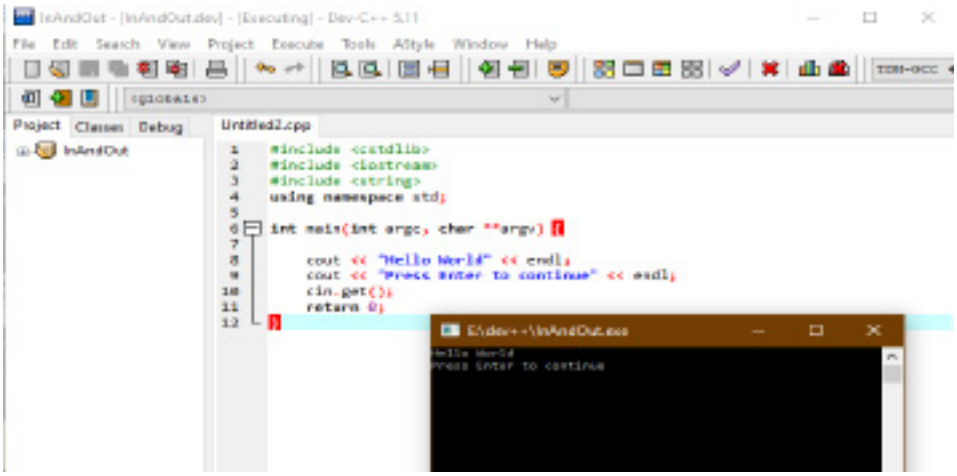
otroCoche->frenar();

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

En programación, las **entradas** y las **salidas** (E/S o I/O del original en inglés input/output) hacen referencia a la colección de señales enviadas y recibidas a lo largo de un programa informático.

Mientras que las entradas son señales que transmiten información hacia la aplicación, las salidas son las señales de respuesta. Dicha información de salida puede ser visualizada por el usuario a través de interfaces determinadas como la **consola**.

La entrada de información se producirá en modo consola por el teclado y la salida mediante una ventana modo texto. Las **consolas** son herramientas de control que permiten la comunicación entre el usuario y el equipo a lo largo de la ejecución de un programa En C++ se utilizan los objetos cin y cout, y en Java los objetos System.in y System.out.



Información de entrada y salida mostrada por la consola.

3.6. Utilización de métodos estáticos

Cuando creamos una instancia de un objeto, estamos creando una representación concreta de ese objeto en el código del programa. Todas las instancias de ese objeto compartirán la misma estructura de métodos y propiedades definidos en su clase.

Sin embargo, puede haber casos en los que necesitemos que ciertos métodos o propiedades sean específicos de la clase misma en lugar de ser específicos de cada instancia individual. En esos casos, podemos utilizar métodos y propiedades estáticos.

3.6.1. Métodos estáticos

Los métodos estáticos son métodos que pertenecen a la clase en lugar de pertenecer a una instancia específica de la clase. Se definen utilizando la palabra clave "static" en la declaración del método. Los métodos estáticos se caracterizan por lo siguiente:

- No requieren que se cree una instancia de la clase para ser invocados.
- Se pueden llamar directamente desde la clase utilizando el nombre de la clase, sin necesidad de crear un objeto.
- No pueden acceder directamente a las propiedades de instancia de la clase, ya que no están asociados a una instancia en particular.

Para invocar un método estático, se utiliza el nombre de la clase seguido del operador de acceso ".", y luego el nombre del método. No es necesario crear una instancia de la clase para llamar a un método estático.

La invocación de métodos estáticos se realiza a nivel de clase, en lugar de a nivel de objeto.

Por ejemplo, en Java:

```
public class Calculadora {
    public static int sumar(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int resultado = Calculadora.sumar(5, 3);
        System.out.println("El resultado de la suma es: " + resultado);
    }
}
```

3.6.2 Propiedades estáticas

Las **propiedades estáticas** son variables que pertenecen a la clase en sí, en lugar de pertenecer a instancias individuales de la clase. Se definen utilizando la palabra clave "static" en la declaración de la propiedad. Estas propiedades son compartidas por todas las instancias de la clase y tienen el mismo valor en todas ellas.

Las propiedades estáticas se acceden utilizando el nombre de la clase seguido del operador de acceso ".", y luego el nombre de la propiedad estática. No es necesario crear una instancia de la clase para acceder a las propiedades estáticas.

Por ejemplo, en C#:

```
public class MiClase {
   public static string nombre = "Ejemplo"; // Propiedad estática
}
public static void Main(string[] args) {
  string nombreClase = MiClase.nombre; // Acceso a la propiedad estática desde la clase
   Console.WriteLine("Nombre de la clase: " + nombreClase);
    }
```

En este ejemplo, se accede a la propiedad estática "nombre" de la clase "MiClase" directamente desde la clase sin crear una instancia. El valor de la propiedad estática se asigna a la variable "nombreClase" y se imprime en la consola.

```
1   class Alumno {                          public class Alumno {
2       char nombre[100];                       String nombre;
3       int edad;                               int edad;
4       char sexo; // 'v' p 'm'                 char sexo; // 'v' p 'm'
5       static int numAlumnos;                  static int numAlumnos=0;
6
7       Alumno(){                               Alumno(){
8           numAlumnos++;                           numAlumnos++;
9       }                                       }
10  };                                       }
11  int Alumno::numAlumnos = 0;
12
```

Ejemplo de la clase Alumno con una variable estática en C++ (izquierda) y Java (derecha).

3.5.3 Diferencias entre métodos estáticos y de instancia:

Los **métodos estáticos y de instancia** tienen diferencias clave en su naturaleza y forma de uso.

Un método estático pertenece a la clase en sí y se puede invocar directamente desde la clase sin la necesidad de crear una instancia del objeto. Estos métodos no están asociados a una instancia particular y no pueden acceder directamente a las propiedades de instancia de la clase. Los métodos estáticos se utilizan comúnmente para implementar funcionalidades que no dependen de un estado específico del objeto, como utilidades o funciones matemáticas.

Por otro lado, los métodos de instancia son específicos de cada objeto creado a partir de la clase. Se invocan en un objeto particular y pueden acceder y modificar las propiedades de instancia asociadas a ese objeto. Estos métodos se utilizan para implementar funcionalidades que interactúan con el estado y comportamiento específico de un objeto.

La diferencia fundamental radica en la forma de invocación y el acceso a las propiedades de instancia. Mientras que los métodos estáticos se invocan directamente desde la clase y no pueden acceder a las propiedades de instancia, los métodos de instancia se invocan en un objeto específico y tienen acceso a sus propiedades de instancia.

La elección entre métodos estáticos y de instancia depende de la naturaleza de la funcionalidad que se desea implementar. Los métodos estáticos son útiles cuando se necesita implementar funcionalidades independientes del estado de los objetos, mientras que los métodos de instancia son apropiados cuando se desea interactuar con las propiedades y el comportamiento específico de un objeto.

3.7. Parámetros y valores devueltos

Denominamos **parámetros** a los datos usados por un método para realizar las acciones para las que ha sido definido. La recepción de estos datos le permitirá devolver un valor resultante que podrá ser almacenado en una variable o utilizado en una expresión. Tomemos como ejemplo un objeto que realiza el cálculo del área de una circunferencia.

Una vez instanciado el **objeto en el código**, podremos generar su método areaCirculo y enviarle los parámetros necesarios (en este caso, el radio) con el fin de obtener el resultado deseado.

La **sintaxis** para crear un objeto en Java y usar el método de manera adecuada sería la siguiente:

```
CalculoArea calculoArea = new
CalculoArea(); float area = calculoArea.areaCirculo(10);
```

Debemos señalar que los parámetros se envían a través de variables definidas entre paréntesis, que el método recoge y asigna adecuadamente para la obtención del resultado. La instrucción return devolverá al código principal el valor del área resultante y quedará almacenado en la variable definida como area. Los parámetros pueden ser pasados a un método de dos maneras diferentes:

- **Por valor.** El parámetro pasado es un valor literal o el valor de una variable. El valor original de la variable en ningún caso se verá alterado por el método.
- **Por referencia.** El parámetro pasado al método es una variable. El valor de esta variable puede ser modificado desde el método.

En lenguaje C++, si no se indica lo contrario un array se pasa por referencia y una variable se envía por valor, como en el siguiente ejemplo:

```
tipo Metodo(tipo VariablePasadaPorValor simple) {
// instrucciones del método
}
```

Para pasar una variable por referencia, a no ser que sea un array, que siempre y por defecto se pasa por referencia, sólo se necesita poner el caracter "&" justo después del nombre de la variable:

```
tipo Metodo(tipo &VariablePasadaPorReferencia) {
    // instrucciones del método
}
```

En Java, no hay una simbología especial para indicar el tipo de parámetro, es decir, si es por valor o por referencia, sólo las normas propias del lenguaje:

1. Los objetos se pasan por referencia.
2. Las referencias a objetos y los tipos elementales por valor.

Puedes observar que un método se define de manera parecida a una varia-ble, indicando primero el tipo de datos que devuelve.

3.8. Librerías de objetos

Una librería de objetos es un fichero que contiene objetos útiles para di-ferentes programas. Estos objetos pueden ser **importados** por nuestros programas, ahorrándonos así el trabajo de tener que crearlos desde cero cada vez que los necesitemos.

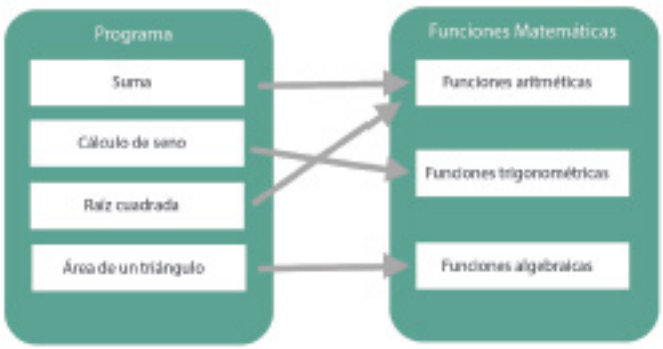
Cuando se crea una librería de objetos se piensa en la resolución de pro-blemas específicos. Así, por ejemplo, podemos tener librerías para resolver problemas matemáticos complejos, para controlar un robot industrial, para dibujar gráficos tridimensionales en la pantalla, o cualquier otra cosa que se nos ocurra.

Los **lenguajes de programación** suelen incluir múltiples librerías para pro-pósitos más o menos generales que no están contemplados en el lenguaje estándar.

Podemos hacer nuestras propias librerías, que nos servirán para reutilizar objetos interesantes y para mantener los objetos de un programa grande bien organizados.

También encontramos librerías comerciales, que nos son ofrecidas para resolver problemas complejos y específicos.

Siguiendo los ejemplos expuestos hasta ahora, podríamos definir una libre-ría que englobara distintas funciones matemáticas según su tipo



Relación entre el programa principal y una librería de objetos de funciones matemáticas.

3.9. Constructores

Un **constructor es un método** especial en la programación orientada a objetos que desempeña un papel fundamental al crear objetos. Su función principal es inicializar el objeto recién creado y establecer sus propiedades en un estado inicial coherente y válido.

Cuando se instancia un objeto utilizando el operador "new", el constructor correspondiente se ejecuta automáticamente. Durante la ejecución del constructor, se pueden llevar a cabo diversas acciones, como asignar valores a las propiedades del objeto, realizar verificaciones o configurar su estado inicial.

El **constructor** se define dentro de la clase y tiene el mismo nombre que la clase a la que pertenece. Esto ayuda a identificar y asociar el constructor con la clase específica. Además, el constructor puede aceptar parámetros, lo que permite personalizar la inicialización del objeto proporcionando valores específicos durante su creación.

Es importante destacar que el constructor no devuelve ningún valor explícito, ya que su propósito principal es inicializar el objeto, no realizar cálculos o retornar resultados. Sin embargo, puede contener lógica adicional y realizar acciones más complejas según las necesidades de la aplicación.

La capacidad de tener varios constructores en una clase, utilizando la sobrecarga de constructores, permite diferentes formas de inicializar un objeto según los argumentos proporcionados. Esto brinda flexibilidad y adaptabilidad al momento de crear objetos en base a diferentes escenarios.

3.9.1. Funciones y características principales de los constructores:

La **principal función** de los constructores es garantizar que un objeto recién creado tenga valores iniciales válidos y coherentes para todas sus propiedades.

Cuando se crea un objeto mediante la instanciación de una clase, el constructor correspondiente se ejecuta automáticamente. Durante la ejecución del constructor, se pueden realizar diversas tareas, como asignar valores a las propiedades del objeto, realizar verificaciones de integridad, configurar conexiones con otros objetos o realizar cualquier otra acción necesaria para garantizar la coherencia y validez del objeto.

Es importante destacar que el **constructor** lleva el mismo nombre que la clase a la que pertenece. Esto ayuda a identificar y asociar fácilmente el constructor con la clase específica. Además, el constructor puede aceptar parámetros, lo que permite personalizar la inicialización del objeto proporcionando valores específicos durante su creación. Esto permite una mayor flexibilidad y adaptabilidad al crear objetos según diferentes escenarios y requisitos.

A diferencia de los métodos regulares, el constructor no devuelve un valor explícito. Su función principal es inicializar el objeto y no realizar cálculos o generar resultados. Sin embargo, el constructor puede contener lógica adicional, como la realización de operaciones de inicialización complejas o la invocación de otros métodos, para garantizar que el objeto esté completamente configurado y listo para su uso.

En algunos casos, puede haber varios constructores definidos en una clase utilizando la sobrecarga de constructores. Esto permite tener diferentes versiones del constructor que aceptan conjuntos diferentes de parámetros. De esta manera, se pueden crear objetos utilizando diferentes combinaciones de valores, lo que brinda más flexibilidad en la creación de objetos personalizados.

Además, en el contexto de la herencia, una clase derivada puede invocar el constructor de su clase base utilizando la palabra clave "super". Esto permite reutilizar la lógica de inicialización de la clase base y extenderla según sea necesario en la clase derivada. De esta manera, se puede establecer un flujo de inicialización coherente y eficiente en toda la jerarquía de herencia.

3.10. Destrucción de objetos y liberación de memoria

La **destrucción de objetos** es uno de los aspectos más importantes a tener en cuenta cuando diseñamos un programa informático. Igual que las variables y todos los otros tipos de almacenamiento de datos, cuando definimos un objeto, éste pasa a ocupar una parte de la memoria de nuestro ordenador. La destrucción de un objeto implica que el objeto deja de existir en el programa y, por consiguiente, deja de estar almacenado en la memoria de nuestra máquina.

Cuando el **código es sencillo**, es posible que éste no sea un aspecto especialmente visible, lo que no significa que deje de ser importante; pero, cuando hablamos de programas complejos, resulta imprescindible. Toda la memoria que podamos ahorrar a la hora de ejecutar un programa, contribuirá a que éste funcione de una manera más ágil en el ordenador.

Si no tenemos en cuenta la liberación de memoria durante la confección de nuestro código, corremos el riesgo de que el programa llegue a colapsarse en el momento de su ejecución. Para evitar esto, los lenguajes de programación incluyen ciertas formas que nos ayudarán, de manera sencilla, a liberar la memoria de aquellos datos que ya no utilizamos.

En Java, la destrucción de objetos se realiza automáticamente mediante el garbage collector, o recolector de basura, que efectúa la máquina virtual. Sin embargo, Java ofrece la posibilidad de implementar el método finalize de la clase del objeto para ejecutar ciertas acciones antes de que se elimine por completo.

```
protected void finalize() {
    // Código a ejecutar antes de liberar el objeto
```

4. Resumen

Los objetos son, sin duda, una de las técnicas de programación más importantes a tener en cuenta a la hora de diseñar y escribir nuestros programas. Las características de los objetos se denominan propiedades, que son variables de cualquier tipo de dato, incluso otros objetos. El comportamiento de los objetos se establece con los métodos, es decir, funciones contenidas en los propios objetos, con o sin parámetros de entrada y devolviendo un valor o no.

Las propiedades pueden ser utilizadas generalmente con el formato "nombre de objeto. nombre de propiedad".

Los métodos de los objetos, como se ha mencionado, son funciones contenidas en el propio objeto. Como con las propiedades, pueden ser llamados como "nombre de objeto.nombre de método(parámetros)".

El diseño o codificación de los objetos se realiza a través de lo que denominamos clase. Una clase no es más que un tipo de datos donde quedan definidos propiedades y métodos. Cuando creamos un objeto a partir de una clase, se dice que estamos instanciando el objeto, y en ese momento se ejecuta el método constructor de la clase. Si utilizamos un método o propiedad directamente desde la clase, sin instanciar un objeto, se habla de propiedades y métodos está-ticos.

En Java, aparte de las palabras reservadas del lenguaje, se utilizan librerías de clases realizadas por terceros o por nosotros mismos. Las clases de una librería pueden agruparse en conjuntos denominados paquetes. Un ejemplo es la clase System, que contiene los objetos in y out que permiten leer y escribir datos en la consola, respectivamente.

Desarrollo de aplicaciones multiplataforma

MÓDULO:

Programación

UNIDAD:

Uso de estructuras de control

CEAC
FP Oficial

MÓDULO:

Programación

UNIDAD:

Uso de estructuras de control

CEAC FP Oficial

ÍNDICE

1. Introducción

Hasta ahora hemos visto cuáles son los elementos principales que intervienen en la programación y hemos aprendido cómo el uso de datos y objetos nos ayudan a organizar nuestro código.

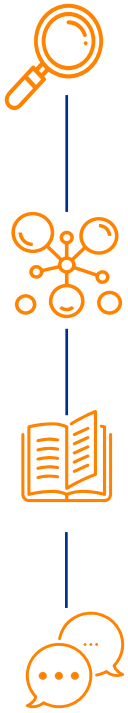
Sin embargo, para la resolución de problemas complejos, es preciso conocer cuáles son las sentencias o acciones que se ejecutan, y en qué momento se ejecutan. Las **estructuras de control**, o sentencias de control, controlan la secuencia o flujo de ejecución de las acciones que realiza nuestro programa.

A lo largo de esta unidad, aprenderemos a controlar los sucesos que desencadenarán las acciones que definiremos en nuestro código durante la ejecución de nuestros programas. La función de las **sentencias de control** es la de dirigir el flujo de las acciones a lo largo de un programa informático con el objetivo de que éste reaccione de un modo u otro según la situación.

Hay estructuras de varios tipos (selección, repetición, salto y excepción). Todas ellas gozan de características y utilidades distintas. Veamos pues, cuáles son y en qué situaciones deberemos usar unas u otras.

2. Objetivos pedagógicos

Los objetivos de aprendizaje que se pretenden alcanzar con esta unidad son:



- **Adquirir conocimientos fundamentales de programación.**
- **Familiarizarse con el lenguaje de programación Java.**
- **Comprender diferentes metodologías de programación.**
- **Desarrollar aplicaciones en Java con diversas funcionalidades.**

3. Uso de estructuras de control

3.1 Estructuras de selección

Las estructuras de selección definen diferentes comportamientos para nuestro programa basándose en la evaluación de sentencias o premisas (condiciones). Estas estructuras permiten controlar el flujo del programa y realizar acciones específicas según las condiciones establecidas.

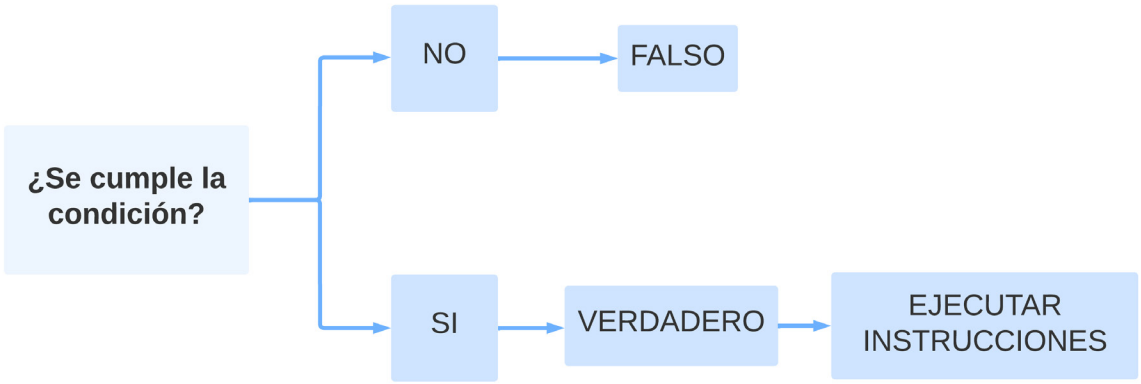
Existen tres tipos:

- **Selección simple.** Su comportamiento se basa en la evaluación de una condición y reacción en consecuencia. Si la premisa se cumple, se desencadenan las acciones definidas; de lo contrario, el programa sigue su curso. La instrucción usada es el condicional if (si).

  Para ejemplificar su uso, podríamos proponer la siguiente situación: imaginemos la página web de una marca de bebidas alcohólicas en la que el acceso está vetado a los menores de 18 años. Cuando un usuario intenta acceder, se le pide la fecha de nacimiento para realizar la comprobación de su edad según el año en curso. Una vez el programa define que el usuario es mayor de 18 años, se le permite el acceso al contenido de la web; de otro modo, seguirá vetado.

- **Selección doble.** Amplía la funcionalidad de las estructuras de selección simple añadiendo un camino alternativo si la premisa evaluada no se cumple. Las instrucciones que las definen son if / else (si / si no).

- **Selección múltiple.** Se basan en la evaluación de colecciones de premisas. El programa elegirá el comportamiento adecuado según sea la premisa que se cumpla. La instrucción que las define se denomina switch (cambia a/selecciona), y los diferentes comportamientos vienen determinados según las sentencias case



Esquema conceptual de una estructura de selección simple.

3.1.1 Estructuras de selección en Java

La sentencia if, o if-else

La **estructura de control if** es una de las principales estructuras de selección utilizadas en programación. Permite ejecutar un bloque de código si se cumple una determinada condición y, opcionalmente, ejecutar un bloque alternativo si la condición no se cumple.

La sintaxis básica del if es la siguiente:

```
if (condición) {
    // bloque de código si la condición es verdadera
}
```

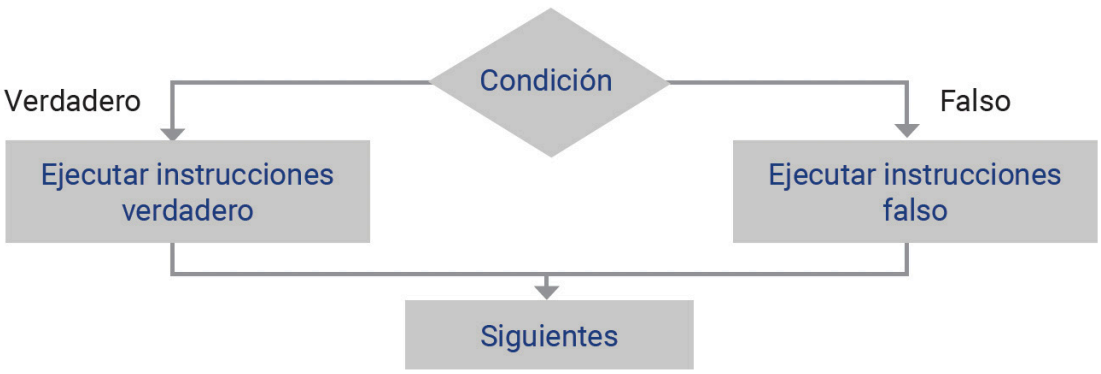
Si la condición especificada dentro de los paréntesis es verdadera, el bloque de código dentro de las llaves se ejecutará. En caso contrario, es decir, si la condición es falsa, el bloque de código se omitirá y la ejecución continuará con las instrucciones siguientes.

La **estructura de control if-else** permite ejecutar un bloque de código si se cumple una condición y otro bloque de código si la condición no se cumple.

Añade a la sentencia if un bloque alternativo utilizando la estructura else, que se ejecutará si la condición del if es falsa. La sintaxis es la siguiente:

```
if (condición) {
    // bloque de código si la condición es verdadera
} else {
    // bloque de código si la condición es falsa
}
```

La expresión lógica de la sentencia if-else tiene que producir un resultado lógico, es decir, true o false. La sentencia puede ser simple, acabada en punto y coma, o puede ser compuesta, siempre encerrada por llaves. Aunque sólo sea necesario el uso de las llaves cuando haya más de una sentencia, se recomienda usarlas siempre para aumentar la legibilidad.



Esquema conceptual de una estructura de selección simple con doble bifurcación.

La selección múltiple, sentencia switch –case:

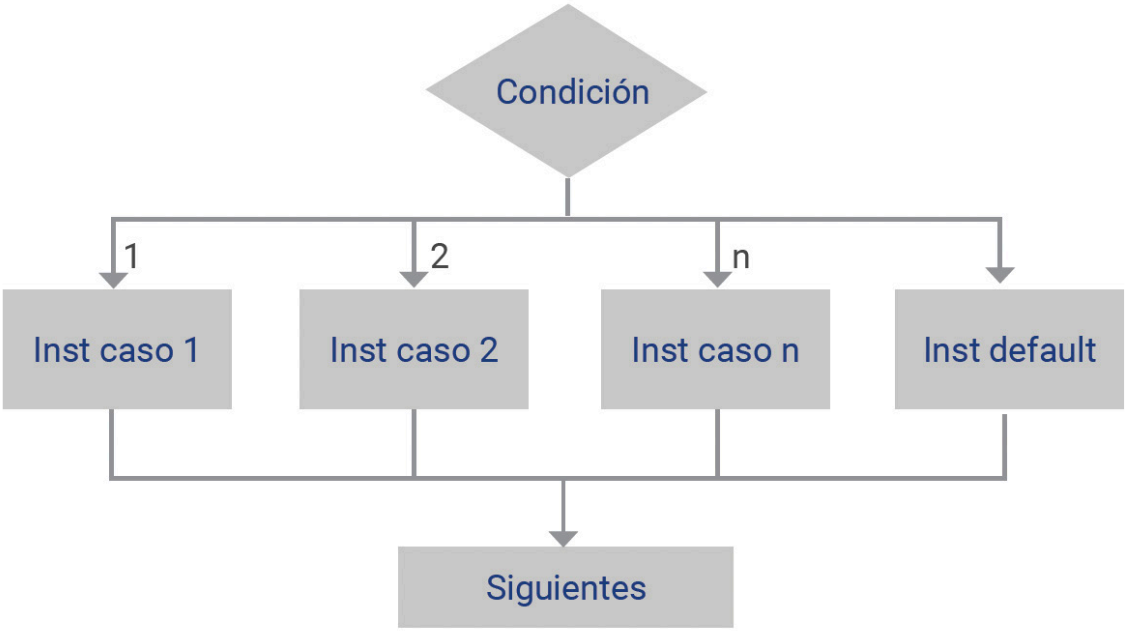
La **estructura de selección switch-case** es una forma de tomar decisiones en base a múltiples opciones. Permite evaluar el valor de una expresión y ejecutar diferentes bloques de código según el valor evaluado. La sintaxis general del switch-case es la siguiente:

```
switch (expresión) {
   case valor1:
      // bloque de código a ejecutar si la expresión es igual a valor1
      break;
   case valor2:
      // bloque de código a ejecutar si la expresión es igual a valor2
      break;
   case valor3:
      // bloque de código a ejecutar si la expresión es igual a valor3
      break;
   // más casos...
   default:
      // bloque de código a ejecutar si ninguno de los casos anteriores coincide con la expresión
      break;
}
```

La expresión, que puede ser una variable, debe dar como resultado int o char.

Cada case acaba en un break y esto hace que, después de cada evaluación y ejecución, se vaya al final. No es obligatorio, pero se suele utilizar para que cada case sea exclusivo. Si no se pusiera, entraría en los demás case, aunque no tuviera ese valor.

Al final, si hay un default, el código de dentro se ejecutará cuando no se haya encontrado coincidencia anterior



Esquema conceptual de una estructura de selección múltiple.

3.2 Estructuras de repetición

Las estructuras de repetición, también conocidas como bucles, son utilizadas en programación para ejecutar un bloque de código repetidamente mientras se cumpla una determinada condición. Estas estructuras permiten automatizar tareas que requieren iteración o repetición de acciones.

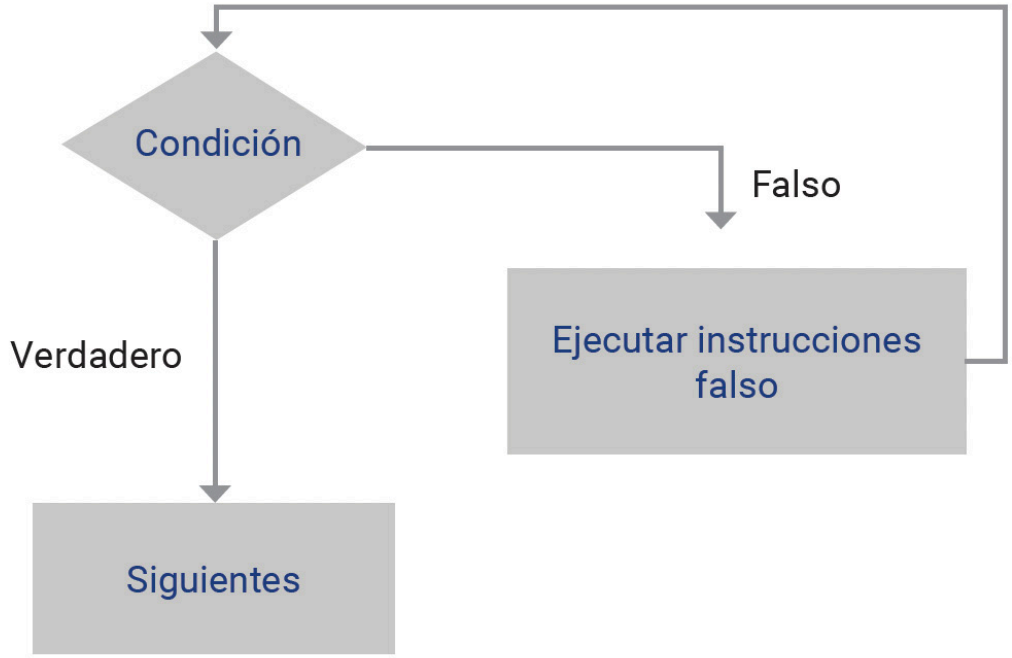
Existen de varios tipos:

- **for.** Determina un número máximo de repeticiones durante las que se ejecutará una acción definida. Cuando se alcanza el número de repeticiones indicado, el bucle acaba y pasa a la siguiente línea de código. Así, permite definir una variable de control, una condición y un incremento/decremento en una sola línea. Es muy útil cuando sabemos que se debe repetir algo "N" veces y nos va bien que un contador se vaya incrementando o decrementando.

- **while**. Su funcionamiento es parecido al de la sentencia **for.** El programa ejecuta una acción mientras se cumpla una condición definida. Si la condición es falsa, el bucle se termina y la ejecución continúa con las instrucciones siguientes. Para que el bucle se rompa, deberá existir en el interior de las acciones definidas una instrucción que convierta la condición en falsa. De otro modo el programa entraría en un bucle infinito y se bloquearía.

- **do while.** Muy similar a la sentencia while, la diferencia es que la comprobación de la premisa se realiza posteriormente a las acciones definidas, es decir, las acciones se ejecutarán como mínimo una vez y la condición se evalúa al final de cada iteración.

Todas las estructuras tienen una utilidad y función según la situación; sin embargo, son las estructuras for y while las más usadas de todas ellas.



Esquema conceptual de una estructura de repetición.

3.2.1 Estructuras de repetición en Java

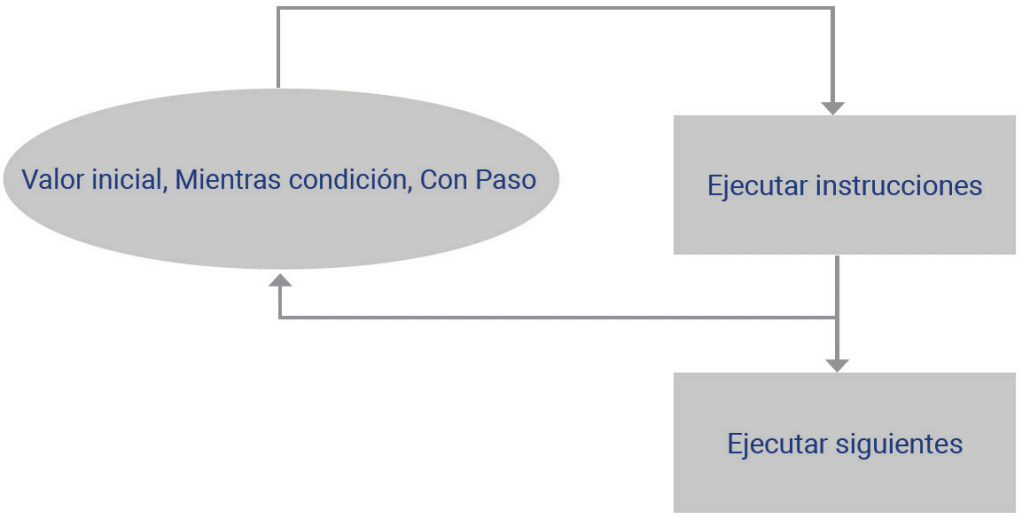
La sentencia for:

El bucle for es una estructura de repetición más completa y versátil. Permite definir una variable de control, una condición y un incremento/decremento en una sola línea. La sintaxis general es la siguiente:

```
for (inicialización; condición; incremento/decremento) {
   // bloque de código a repetir
}
```

La sentencia **for** ejecuta un grupo de instrucciones repetidamente con las siguientes restricciones (separadas por punto y coma):

- **Inicialización**. Una expresión que inicializa la variable o variables que se utilizan en la iteración. Por ejemplo, i = 0.
- **Finalización**. Mientras se cumplan las condiciones que se incluyen, se irán ejecutando repetidamente las instrucciones contenidas en el **for.** Por ejemplo, i < 5.
- **Iteración**. Incremento o decremento de la variable o variables que a cada iteración se va a actualizar. Por ejemplo, i++ hará que a cada vuelta se incremente en 1 la variable i, pero también podría ser i--, o i = i +2.



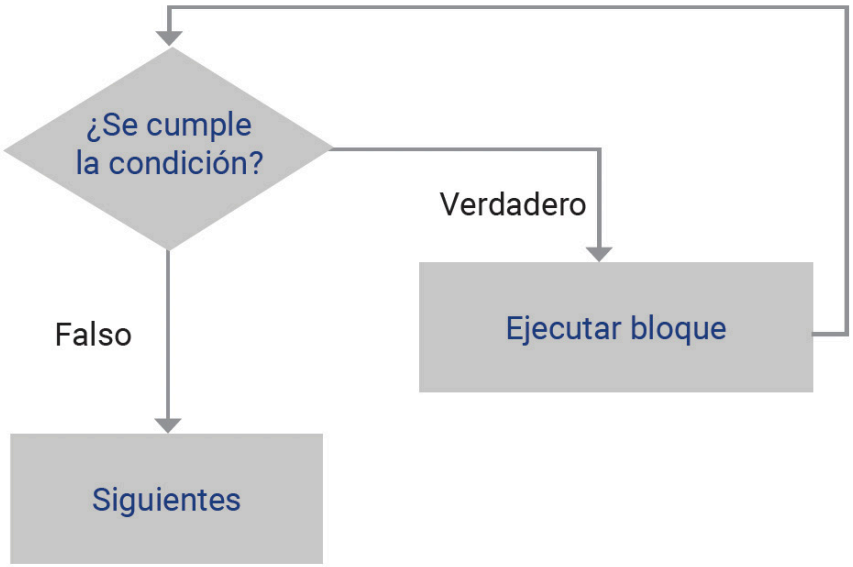
Esquema conceptual de una estructura for

La sentencia while:

El bucle while repite un bloque de código mientras se cumpla una condición específica. La sintaxis básica es la siguiente:

```
while (condición) {
   // bloque de código a repetir
}
```

Esta estructura ejecuta repetidamente las sentencias que encierra mientras la expresión condicional sea verdadera cada iteración, se evaluará la condición y, si se cumple, se ejecutará el bloque de código. Si la condición es falsa, el bucle se termina y la ejecución continúa con las instrucciones siguientes.



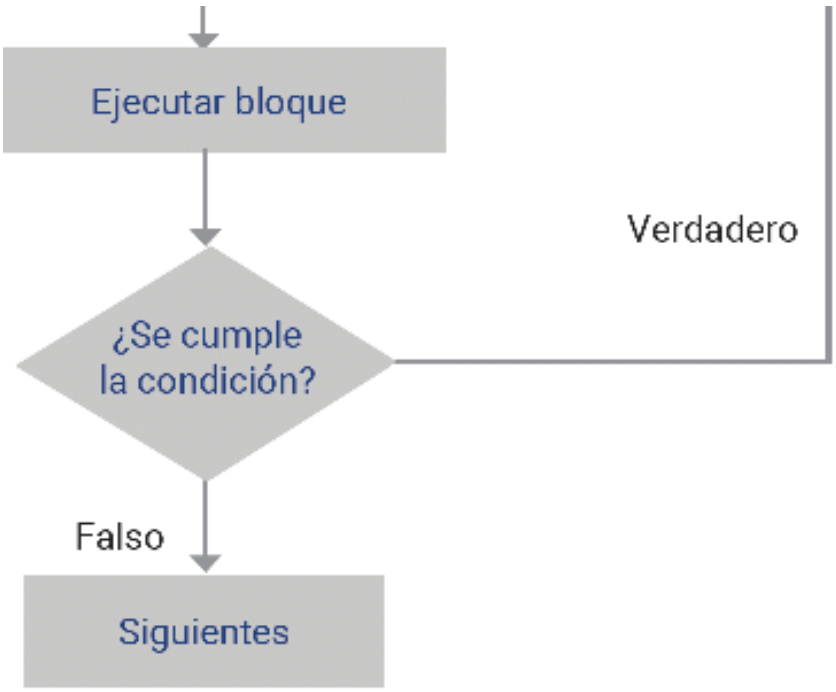
Esquema conceptual de una estructura while.

La sentencia do - while:

El bucle do-while es similar al bucle while, pero se diferencia en que la condición se evalúa al final de cada iteración. Esto significa que el bloque de código se ejecutará al menos una vez, incluso si la condición es falsa desde el principio. La sintaxis es la siguiente:

```
do {
   // bloque de código a repetir
} while (condición);
```

Esta estructura ejecuta repetidamente las sentencias que encierra mientras la expresión condicional sea verdadera. Sin embargo, como la condición está al final, a diferencia del while. El bloque de código se ejecuta primero y luego se verifica la condición. Si la condición se cumple, se vuelve a ejecutar el bloque de código. Si la condición es falsa, el bucle se termina.



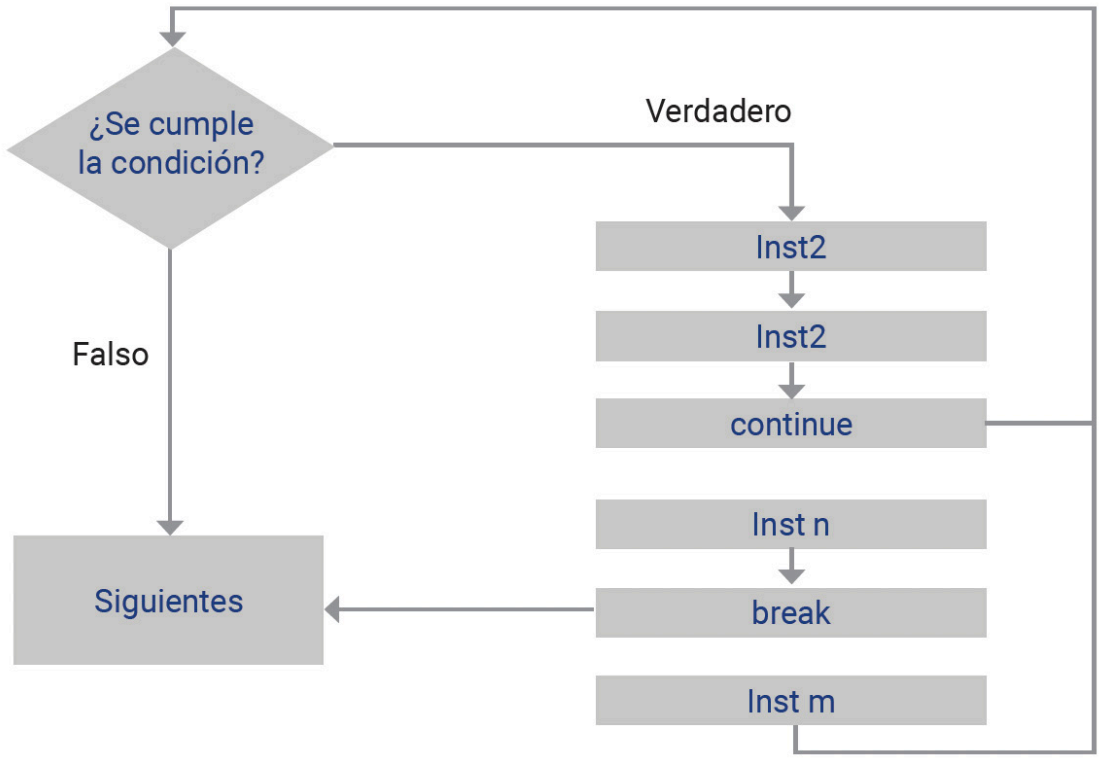
Esquema conceptual de una estructura do-while.

3.3 Estructuras de salto

Las estructuras de salto sirven para interrumpir el flujo normal del programa y desviarlo hacia un punto diferente. Podemos destacar tres tipos:

- **break**. Provoca una salida brusca de un bucle for, while o do-while. También en la sentencia switch, permite que los case sean exclusivos. Tras ella, continúa la ejecución del resto del programa.
- **continue**. Interrumpe la ejecución normal de un bucle. ir directamente a la siguiente repetición, pero sin finalizar el bucle que continua su proceso.
- **return**. Sólo se permite su uso dentro de funciones o bloques de instrucciones. Provoca la salida inmediata del bloque y retorna al punto desde donde había sido activado, devolviendo, si es necesario, un valor.

Es importante no usar estas acciones más que en las situaciones definidas. Cualquier otro uso forzado podría provocar el colapso del programa.



Esquema conceptual de las diferentes estructuras de salto y sus usos permitidos.

3.4 Control de excepciones

El control de excepciones es una estructura que se utiliza para evitar que un programa provoque un error y se cierre cuando intenta ejecutar un código no válido o que no pueda interpretar.

Una excepción representa un evento anormal o inesperado que interrumpe el flujo normal del programa. El control de excepciones permite detectar y manejar estas excepciones de manera controlada, evitando que el programa se detenga abruptamente.

PARA SABER MÁS: Las estructuras de control son combinables entre sí y, aunque no es aconsejable abusar de ello, si la ocasión lo requiere, pueden incluso anidarse las unas con las otras para crear nuevas estructuras.

El **bloque try-catch** es una estructura utilizada en muchos lenguajes de programación para manejar excepciones. Está compuesto por:

- **Bloque try.** se definen las sentencias a ejecutar y que pueden producir un error. En caso de éxito, el programa sigue su flujo normal, en caso contrario pasa a la siguiente instrucción catch.
- **Bloque catch.** Bloque en el que estarán las instrucciones que gestionarán el error generado.

Como ejemplo de uso para una estructura de control de excepciones podríamos tomar una función cuyos parámetros serán usados para realizar una división. Si al parámetro definido como divisor se le asignara el valor 0,el programa podría colapsarse al no poder realizar la operación. Gracias al control de excepciones el programa podrá determinar que ha ocurrido un error, actuar en consecuencia, advertir de lo ocurrido al usuario y seguir su curso normal.

Esquema conceptual del proceso de control de excepciones en una función en C++.

```
int division(int x, int y) {
    try {
        return (x / y);
    } catch (const std::exception& e) {
        throw std::runtime_error("Error: Los parámetros de la división no son evaluables. " + std::string(e.what()));
    }
}
```

3.4.1 Excepciones en Java

```
try {
   // <lo que sea>;
} catch (Exception& e) {
   // <lo que sea>;
}
```

Al ejecutar las **sentencias incluidas en el try**, si se produce un error, Java creará un objeto de la clase Exception o de una de sus heredadas. Este objeto será pasado al catch y se ejecutarán las sentencias que se incluyen en el mismo.

Se pueden incluir varias cláusulas catch con distintos tipos de excepciones. Siempre se deben poner de mayor a menor detalle. Es decir, como la clase Exception es de donde heredan todas, ésta siempre irá la última y, por lo tanto, el grupo de sentencias de su catch siempre se ejecutará si no se ha cazado en una excepción más particular de un catch anterior del mismo try–catch.

RECUERDA: Las estructuras de salto están diseñadas para ser introducidas en ciertos puntos concretos del código en los que se requieren. Su uso fuera de esos puntos no es aconsejable, salvo que no se halle una mejor solución.

Además de capturar y manejar **excepciones**, en algunos casos puede ser necesario lanzar una excepción explícitamente. Esto se hace utilizando la palabra clave throw, seguida de una instancia de excepción. Ejemplo de lanzamiento de excepción en Java:

```
public int dividir(int dividendo, int divisor) throws ArithmeticException {
   if (divisor == 0) {
      throw new ArithmeticException("División por cero");
   }
   return dividendo / divisor;
}
```

En este ejemplo, si el divisor es cero, se lanza una excepción ArithmeticException con un mensaje descriptivo.

Por otro lado, cabe mencionar la existencia de la cláusula finally, que se utiliza en conjunto con los bloques try-catch para especificar un bloque de código que se ejecutará siempre, independientemente de si se produjo o no una excepción.

El **código dentro del bloque finally** se ejecuta después de que se haya manejado la excepción o después de que se haya propagado la excepción.
Ejemplo de uso de la cláusula finally:

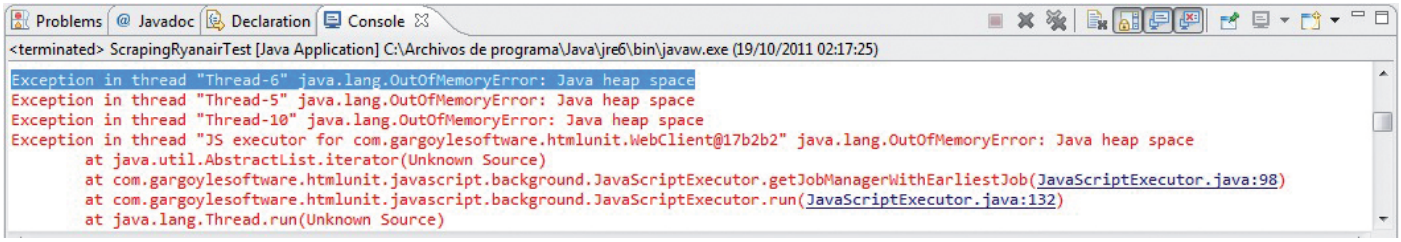
```
try {
    // Código que puede lanzar una excepción
    // ...
} catch (Exception e) {
    // Manejo de la excepción
    // ...
} finally {
    // Código que se ejecuta siempre, incluso si se lanzó una excepción
    // ...
}
```

En este ejemplo, el código dentro del bloque finally se ejecutará sin importar si se produjo una excepción o no. Es útil para realizar tareas de limpieza o liberación de recursos que deben realizarse en cualquier caso.

3.5 Prueba y depuración

Una vez hemos acabado de escribir el código de nuestro programa llega el momento de someterlo al proceso de **prueba y depuración**. La finalidad de las pruebas es definir si las acciones que realiza el programa durante su ejecución dan como resultado la resolución del problema para el que ha sido diseñado.

Si los resultados obtenidos no son satisfactorios deberemos revisar nuestro código, detectar en qué puntos se producen errores y depurarlos. La prueba y la depuración son procesos que deben hacerse tanto a lo largo de la escritura del programa, para comprobar la funcionalidad de **pequeños bloques de código**, como al finalizar la aplicación para comprobar su funcionalidad con respecto a los usuarios. Es en este proceso donde toma mayor importancia la consola de errores, ya que nos ayudará a detectar los puntos donde se producen errores y conocer de qué tipo son.



Listado de errores mostrados por la consola durante la ejecución de un programa en Java.

3.6 Documentación

La documentación es la guía o comunicación escrita que ayuda a comprender el uso de un programa y facilita sus **futuras modificaciones** (mantenimiento). Debe recoger todos los elementos y material creado en las diferentes fases del desarrollo, además de las normas de instalación y recomendaciones para su ejecución. La documentación se puede dividir en:

- **Documentación técnica:** Describe la arquitectura, el diseño y la implementación del software, incluyendo diagramas, especificaciones técnicas, APIs y otros detalles relacionados con la implementación del sistema.
- **Documentación de usuario**: Proporciona información para los usuarios finales sobre cómo instalar, configurar y utilizar el software de manera efectiva, a través de manuales, guías de usuario, tutoriales y ejemplos.
- **Documentación de requisitos**: Describe los requisitos funcionales y no funcionales del software, incluyendo casos de uso, especificaciones de requisitos y documentación de diseño.
- **Documentación de pruebas**: Documenta los casos de prueba, resultados de pruebas y procedimientos relacionados con las pruebas del software, para asegurar su calidad y correcto funcionamiento.

RECUERDA: La documentación es uno de los procesos que suelen pasarse por alto; sin embargo, supone el complemento perfecto para un producto final usable y estable.

4. Resumen

Como cualquier lenguaje de programación, Java debe implementar las estructuras de control que permiten cambiar la ejecución secuencial del programa y, en definitiva, nos ayudan al implementar cualquier tipo de algoritmo.

Las estructura de control de selección, if – else , ejecuta las instrucciones contenidas en el if siempre que se cumpla la condición; en otro caso, ejecutará las del else, siempre que se haya explicitado esta palabra clave.

Las estructuras de control switch – case evalúa el valor de una variable y ejecuta las instrucciones que se encuentran en el case correspondiente al valor.

Las estructuras de repetición son estructuras de control que ejecutan un conjunto de instrucciones repetitivamente. En la estructura while, se ejecuta el conjunto mientras se cumpla la condición. La estructura do – while, ejecuta primero el conjunto de instrucciones y después evalúa, repitiendo el proceso mientras se cumpla la condición. La estructura for (inicialización;condicion es;incremento) ejecuta el conjunto inicializando unas variables, mientras se cumplan las condiciones e incrementando o decrementando las variables de la parte del incremento.

En Java, se puede prever un error y ejecutar unas instrucciones cuando ocurre. Este proceso se denomina control de excepciones, y se realiza mediante la estructura try - catch. Cuando las instrucciones dentro del try tienen un error se ejecutarán las instrucciones contenidas en el catch.

Los entornos de programación integrados (IDE, del acrónimo en inglés) tienen utilidades para depurar los programas, es decir, seguirlos paso a paso para encontrar los posibles orígenes de un mal funcionamiento.

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

MÓDULO:
Programación

UNIDAD:
Desarrollo de clases

CEAC
FP Oficial

MÓDULO:
Programación

UNIDAD:
Desarrollo de clases

CEAC
FP Oficial

ÍNDICE

4. DESARROLLO DE CLASES

En programación, el concepto de **clase** (*class*) está directamente relacionado con el concepto de **objeto** (*object*).

Se define una clase como el concepto bajo el que se agrupan objetos de un mismo tipo.

Si tomamos un objeto como la representación abstracta de un concepto real, una clase sería aquello que define de qué tipo de objeto se trata. Si tomamos el ejemplo de un coche, como concepto tangible, podríamos representarlo según unas características propias: su color, la velocidad máxima que puede alcanzar, su cubicaje, etc. Sin embargo, si tomamos el ejemplo de una motocicleta, veremos que, salvando las diferencias, muchas de las características que podemos asignarle coinciden con las de un coche. Ambos poseen velocidad máxima y cubicaje, color, marca, ruedas, en definitiva, podríamos afirmar que ambos pertenecen a un mismo grupo: el de los vehículos.

Una clase es, pues, el concepto bajo el cual se engloban los objetos que comparten una amplia serie de características comunes. La clase vehículo sería el grupo principal de toda una serie de objetos que constituyen diversas subclases, en ese caso: coches, motocicletas, camiones, furgonetas, etc. Como objeto principal, la clase vehículo contendría en sí misma todas las características compartidas por aquellos objetos que derivan de ella. Y, en tanto que objetos derivados, un coche o una motocicleta tendrían, además de sus características comunes, las suyas propias.

Las clases son uno de los aspectos clave de la **Programación Orientada a Objetos (POO).** Sobre ellas se basa la estructura y funcionamiento de los programas informáticos. Si dominamos y comprendemos su filosofía, podremos diseñar programas que posean una estructura lógica firme, que contribuirán a un mejor funcionamiento de la aplicación final, así como a una mayor facilidad en el manejo y mantenimiento.

4.1 Estructura y miembros de una clase

En términos generales, una clase se estructura exactamente igual que un **objeto** (Figura 4.1). Recordando la estructura de un objeto, hay un primer bloque para los atributos, llamados aquí **variables miembro**, y un segundo bloque de métodos o acciones. Las variables miembro y los **métodos** definidos en una clase serán compartidos o heredados por todos los objetos que de ella deriven. A las **variables miembro** se puede otorgar la ca-

característica private (privado), que impide que sean modificadas libremente desde un punto externo a la clase. Este concepto responde a un principio de la **POO** llamado **ocultación**, que, indica que para modificar o acceder a una variable propia de una clase, es preciso utilizar un paso intermedio llamado **mensaje**. Los mensajes se usan tanto para acceder y modificar las variables miembro de una clase, como para comunicar clases entre sí. Según el lenguaje de programación, existen varios modos para comunicar clases, pero el más utilizado es a través de los **métodos** especiales (*getters* y *setters*, Figura 4.2) que, mediante instrucciones concretas, permiten modificar y obtener los valores de las variables miembro de una clase. En los siguientes apartados de esta unidad veremos con más detalles algunos de estos conceptos.

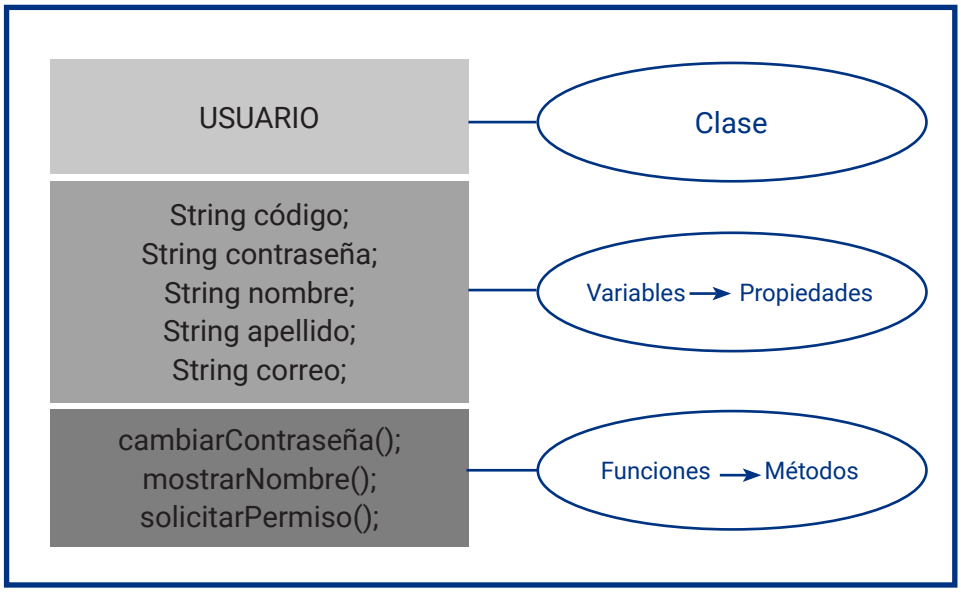


Figura 4.1
Esquema de la estructura básica de una clase y sus diferentes bloques.

```
1   public class Persona {
2       protected String nombre;
3       protected String apellido1;
4       protected String apellido2;
5
6       private int edad;
7
8       public void setEdad(int edad){
9          if(edad > 0) {
10             this.edad = edad;
11         }
12      }
13      public int getEdad(){
14          return (this.edad);
15      }
16  }
17
```

Figura 4.2
Ejemplos de, *setter* y *getter*.

4.1.1 Sintaxis de una clase en Java

```
class Persona{
      //propiedades
      ...
      //métodos
      ...
}
La definición completa de una clase en Java es:
[public] [final | abstract] class Clase [extends ClaseMadre] [im-
plements Interfase1
[, Interfase2 ]…]
```

Los modificadores se irán explicando a lo largo del curso en las situaciones concretas donde se utilicen. Nos quedamos pues, con la declaración simple.

4.2 Creación de atributos

Cuando hablamos de atributos de una clase, nos referimos a todo aquel dato destinado a ser compartido por todos los objetos que de ella deriven. Los atributos pueden ser de tres tipos:

- **Públicos (*public*).** Son aquellos a los que puede accederse o modificarse directamente tanto a nivel interno como externo de la clase.

- **Privados (*private*).** Denominados también **variables miembro**, son aquellos que sólo pueden ser modificados directamente a nivel interno de la clase. Para modificar o acceder a un atributo privado desde un punto externo a la clase a la que pertenece, deberemos utilizar un proceso llamado **mensaje**, el cual establece unos métodos (funciones) específicos para tales efectos.

- **Protegidos (*protected*).** Similar al anterior, pero accesible desde las clases derivadas.

4.2.1 Sintaxis de atributos en Java

Los atributos de una clase en Java son las propiedades del mismo.

```
private | public class Persona{
      //propiedades
      int i=0;
      char letra ='a';
      //métodos
      ...
}
```

Los modificadores se irán explicando a lo largo del curso en las situaciones concretas donde se utilicen. Nos quedamos pues, con la declaración simple. Sin embargo, hay que recordar que una propiedad estática (static) es aquella que es única a nivel de clase, no de objeto.

4.3 Creación de métodos

Los métodos de una clase hacen referencia a las acciones (funciones) comunes a todos los objetos que de ella se deriven. Igual que sus atributos, pueden ser de tipo público (public), privado (private), o protegido (protected):

- **Métodos públicos (*public*).** Un método público puede ser accedido directamente tanto a nivel interno como externo de la clase. Por norma general se utilizan para impartir órdenes a una clase con el fin de que active una de sus funciones.

- **Métodos privados (*private*).** Sólo pueden ser llamados a nivel interno de la clase. Hacen referencia a funciones propias de un objeto, algo que éste no tiene por qué compartir con otras clases u objetos derivados.

- **Métodos protegidos (*protected*).** Similar al anterior, pero accesible desde las clases derivadas

Tal y como hemos visto en el apartado anterior, existen también métodos públicos específicos para acceder a los atributos privados de una clase. Son los llamados getters y setters (Figura 4.3):

```
1   static int nPersonas=0;
2   public Persona(){
3       this.nombre="Desconocido";
4       this.apellido1="Desconocido";
5       nPersonas++;
6   }
7   public Persona(String nom, String ap1){
8       this.nombre = nom;
9       this.apellido1=ap1;
10      nPersonas++;
11  }
12  public void finalize(){
13      nPersonas--;
14  }
15
```

Figura 4.3
Ejemplo en Java de métodos públicos getter y setter para el acceso/modificación de propiedades privadas.

Recuerda

Los métodos definen las funciones que serán capaces de desempeñar nuestras clases a lo largo de la ejecución del programa. Según la estructura con la que organicemos nuestras clases, ésta nos permitirá evitar la repetición de métodos para todas aquellas clases con características comunes.

- ***Getters.*** Sirven para conocer el valor de un atributo en concreto. Es un método de sólo lectura que se limita a devolver el valor de un único dato en concreto y, en ningún caso, podrá modificarlo.

- ***Setters.*** Sirven para asignar valor a un atributo, bien para modificarlo, bien para iniciarlo si éste estaba vacío.

4.3.1 Sintaxis de métodos en Java

Los métodos de una clase en Java son acciones que ofrecerá el objeto. Los modificadores se irán explicando a lo largo del curso en las situaciones concretas donde se utilicen. Nos quedamos, pues, con la declaración simple.

4.4 Creación de constructores

Igual que en la estructura de un objeto, llamamos **constructor** a la función especial de una clase. La finalidad del constructor es inicializar la clase y permitir que esta tenga una identidad única dentro del programa. Los únicos requisitos que hay que tener en cuenta es que se deberá definir como público (public) y su nombre deberá ser exactamente el mismo que el nombre de la clase a la que representa (Figura 4.4).

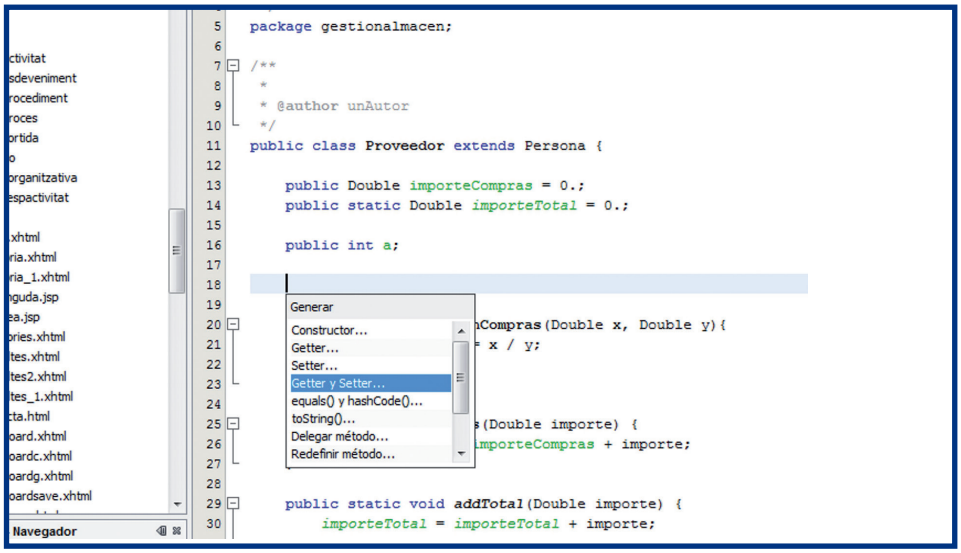


Figura 4.4
Muestra de ayuda del IDE NetBeans para creación de código (getter y setter).

Por norma general, el nombre de las clases se define con la primera letra de su propio nombre en mayúscula. Si el nombre incluye más de una palabra, ésta se definirá también con la primera letra en mayúscula y sin separación con respecto a la anterior. El constructor deberá tener el mismo nombre que la clase.

```
private | public class Persona{
      //propiedades
      int i=0;
      char letra='a';
      //métodos
      public int sumamos(int x1, int x2){
            return x1+x2;
      }
```

```
}

La definición completa de un método de una clase en Java es:
[private|protected|public] [static] [abstract] [final] [native]
[synchronized] TipoDevuelto NombreMétodo ( [tipo1 nombre1[, tipo2
nombre2 ]…] ) [throws excepción1 [,excepción2]… ]
```

4.4.1 Sintaxis de constructores en Java

Los constructores de una clase en Java se diferencian en los parámetros con los que se llaman desde la instrucción ***new***. No devuelven nada y se llaman como la clase.

```
private | public class Persona{
      //propiedades
      int i=0;
      char letra='a';

      //constructores
      //un primer constructor sin parámetros
      public Persona(){
            i=5;
            ...
      }
      //un segundo constructor con parámetros
      public Persona(int i){
            this.i = i;
      }

      //métodos
      public int sumamos(int x1, int x2){
            return x1+x2;
      }
}
```

4.5 Encapsulación, ocultamiento y visibilidad

En este apartado hablaremos de los tres conceptos básicos bajo los que se rigen las clases: la encapsulación, la ocultación y la visibilidad.

- **Encapsulación.** Hace referencia al modo en que se organiza la estructura de una clase. Es el principio que controla el acceso a sus atributos y métodos. Su objetivo es mantener el orden de los componentes de la clase y facilitar su uso.

- **Ocultamiento.** Se trata del concepto de ocultar las funciones y atributos de una clase que sólo se usan a nivel interno y que, por extensión, no tienen necesidad de ser accesibles externamente. Este proceso ayuda a que la clase sea más segura, limitando el acceso a las funciones para las que ha sido diseñada.

- **Visibilidad.** Consiste en definir los parámetros que, a su vez, definirán el ocultamiento de los elementos de una clase. Existen tres niveles de acceso a los atributos y métodos de una clase:

- Público. Cualquier clase puede acceder al elemento.
- Protegido. Sólo las clases hijas o derivadas pueden acceder al elemento.
- Privado. El elemento está reservado únicamente para la clase que lo contiene.

4.5.1 Sintaxis de visibilidad en Java

```
private | public class Persona{
	//propiedades
	public int i = 0; //pública
	private char letra = 'a'; //privada sólo accesible a la clase
	protected double d = 0.; //protegida sólo por clases heredadas
	y  por otras en el paquete
	//constructores
	//un primer constructor sin parámetros
	public Persona(){
		i=5;
		...
	}
	//un segundo constructor con parámetros
	public Persona(int i){
		this.i = i;
	}
	//métodos
	public int sumamos(int x1, int x2){
		return x1+x2;
	}
}
```

4.6 Utilización de clases y objetos

El **uso de clases y objetos** define el proceso a través del cual podremos agrupar, organizar e instanciar elementos en el código de nuestras aplicaciones (Figura 4.5). Lo primero que deberemos tener en cuenta a la hora de crear un objeto es si éste hará referencia a un concepto concreto o si podrá pertenecer a un grupo más amplio de elementos con características comunes. En este caso es cuando toman especial relevancia los conceptos de **encapsulación** y **empaquetado** de clases. Si nuestro objeto representa un elemento conciso o aislado, él mismo representará a su clase; de lo contrario, estaríamos hablando de un objeto derivado de un concepto más general que formaría parte de un grupo más amplio de elementos. En este caso se le atribuye al objeto la categoría de **subclase**.

En cuanto al método de uso en el código de clases y objetos es el mismo, la **instanciación**. Recordemos que la **instanciación** es el proceso a través del cual podremos generar un objeto en el código y utilizar sus funciones para que intervengan en la ejecución de nuestro programa. Una vez el objeto o la clase están instanciados en el código podremos proceder a aplicar sus métodos y propiedades para que intervengan en el curso del programa, bien sea de manera directa, o bien entre ellos a través de mensajes.

Figura 4.5
La utilización de clases y objetos debe tener en cuenta diversos conceptos.

Recuerda

El encapsulamiento y la visibilidad de las clases creadas y los elementos definidos en ellas son un factor clave para el buen funcionamiento de nuestro programa.

4.6.1 Sintaxis de utilización de clases y objetos en java

```
class Persona{
      //propiedades
      public int i = 0; //pública
      private char letra = 'a'; //privada sólo accesible a la
clase
      protected double d = 0.; //protegida sólo por clases
      heredadas y por otras en el paquete
      //constructores
      //un primer constructor sin parámetros
      public Persona(){
            i=5;
            ...
      }
      //un segundo constructor con parámetros
      public Persona(int i){
            this.i = i;
      }
      //métodos
      public int sumamos(int x1, int x2){
            return x1+x2;
      }
}
class Coche{
      int k = 0;
      Persona conductor = new Persona(); // creación del objeto
      condutor a partir de la clase Persona
      k = conductor.sumamos(3,4); // utilización del método
      sumamos del objeto conductor
      …

}
```

4.7 Utilización de clases heredadas

En **POO**, la **herencia** es el mecanismo fundamental que permite la reutilización y extensibilidad de un programa. A través de ella, los diseñadores pueden construir nuevas clases partiendo de una jerarquía de clases ya existente evitando el rediseño y la modificación de las mismas.

La **herencia** facilita la creación de objetos a partir de otros ya definidos a través de la obtención de características (métodos y atributos) comunes.

Con la **herencia** se establecen relaciones entre las clases generales y sus derivadas o más específicas. Por ejemplo, si declaramos una clase párrafo derivada de una clase texto, todos los métodos y variables asociadas con la clase texto, serán automáticamente heredados por la subclase párrafo.

La **herencia** es uno de los mecanismos más importantes que posee la POO, por medio del cual una clase puede derivarse de otra, de modo que se extienda su funcionalidad (Figura 4.6).

Según el lenguaje de programación, podemos encontrar dos tipos de herencias:

- **Simple.** Una clase puede derivar únicamente de otra clase.
- **Múltiple.** Una clase puede extender las características de una o más clases. Algunos lenguajes, como Java o C#, no permiten este tipo de herencia y otros como C++ sí.

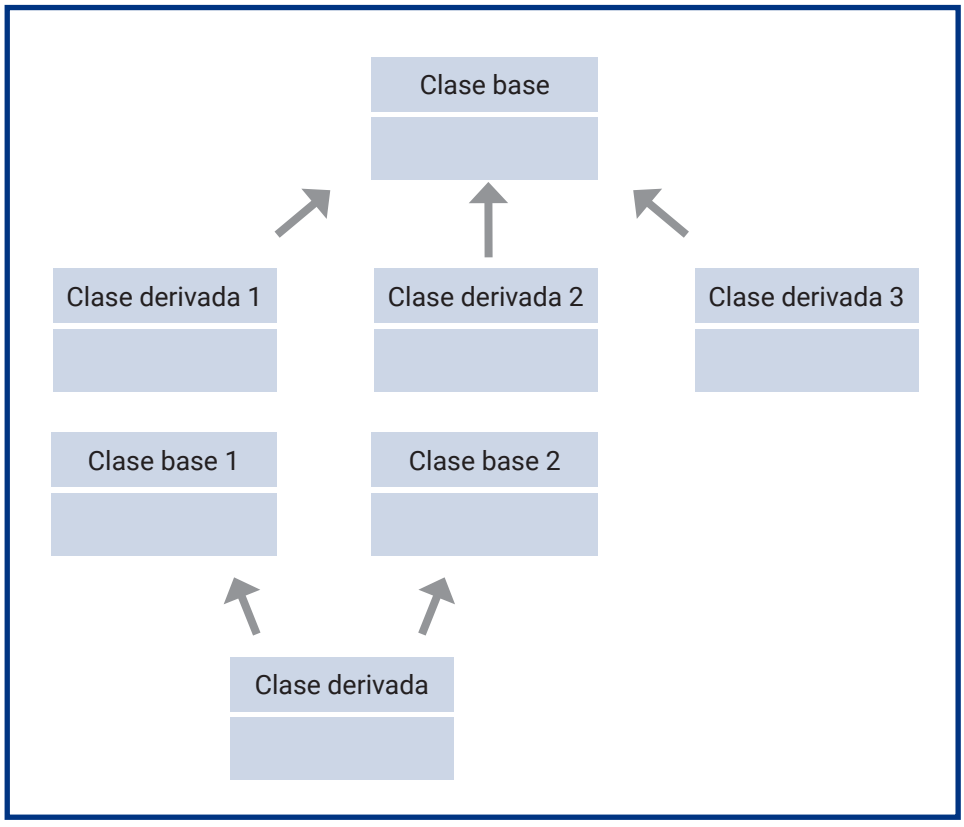


Figura 4.6
Ejemplo de herencia simple (primera) y compuesta (segunda).

4.7.1 Sintaxis de herencia de clases

```
class Persona{
      //propiedades
      public int i = 0; //pública
      private char letra = 'a'; //privada sólo accesible a la clase
      protected double d = 0.; //protegida sólo por clases
      heredadas y por otras en el paquete
      //constructores
      //un primer constructor sin parámetros
      public Persona(){
            i=5;
            ...
      }
      //un segundo constructor con parámetros
      public Persona(int i){
      this.i = i;
      }
      //métodos
      public int sumamos(int x1, int x2){
      return x1+x2;
      }
}
class Alumno extends Persona{
      public double nota = 0;
      public int sumamos(int x1, int x2){
            int a = super. sumamos(x1,x2);
            System.out.println("la suma es:  "+ a}
            return a;
      }
}
```

En este caso, **Alumno** hereda de **Persona**. Por lo tanto, tiene las mismas propiedades y los mismos métodos con las salvedades siguientes:

- Se ha añadido la propiedad **nota**.
- Se ha reescrito el método **sumamos;** que, además de hacer la suma (llamando a la clase de la que hereda con super.sumamos x1, x2), también imprime el resultado.

Recuerda

Agrupando las clases en paquetes, el código resultante del programa podrá ser reutilizado en futuros proyectos y aplicaciones de funcionalidades similares.

4.8 Empaquetado de clases

El empaquetado de clases (class package) es el proceso que permite agrupar las distintas partes de un programa cuya funcionalidad tienen ele-

mentos comunes (Figuras 4.7 y 4.8). Con el uso de paquetes de clases obtendremos las siguientes ventajas con respecto a la programación de nuestras aplicaciones:

- Agrupamiento de clases con características comunes creando una estructura organizativa del código en función de grupos distribuidos según sus funciones.
- Reutilización de código. Al agrupar un conjunto de clases en un paquete, éste podrá ser reutilizado en futuros proyectos, incluso por otros usuarios que quieran desarrollar aplicaciones de funcionalidades similares.
- Mayor seguridad gracias a los niveles de acceso. Se limita el acceso sólo a las acciones ejecutables, no a las que se usan a nivel interno de las clases. Esto evitará que ciertas funciones interfieran entre ellas o que se ejecuten acciones en puntos inadecuados del código fuente.

Para saber más

La herencia entre clases ayuda a definir nuevas clases (subclases) aprovechando las características de una clase más general. Al mismo tiempo, podremos crear también subclases a partir de clases derivadas, lo que nos permitirá alcanzar niveles más específicos para funcionalidades de nuestros objetos.

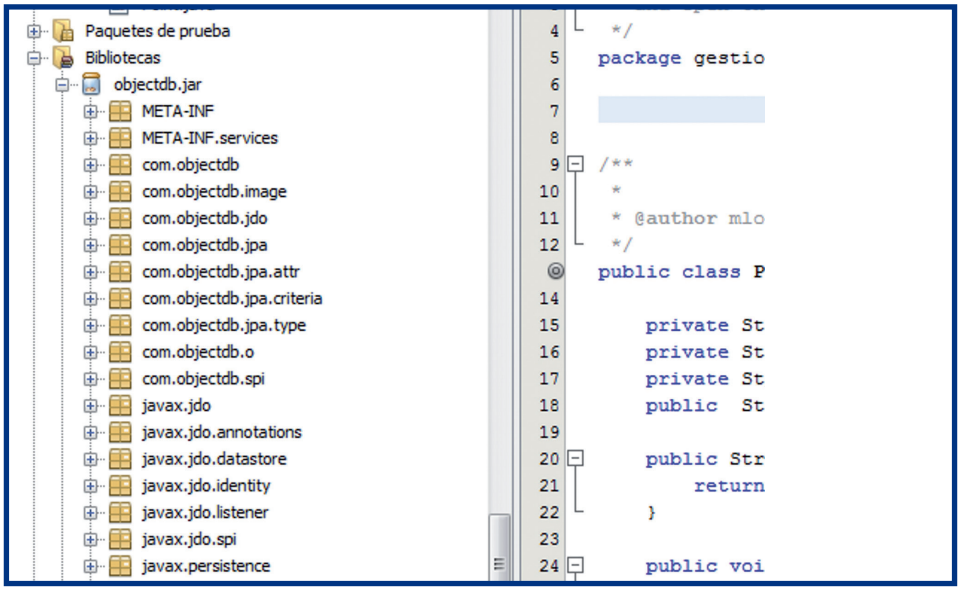


Figura 4.7
Diferentes paquetes de ejemplo en Java.

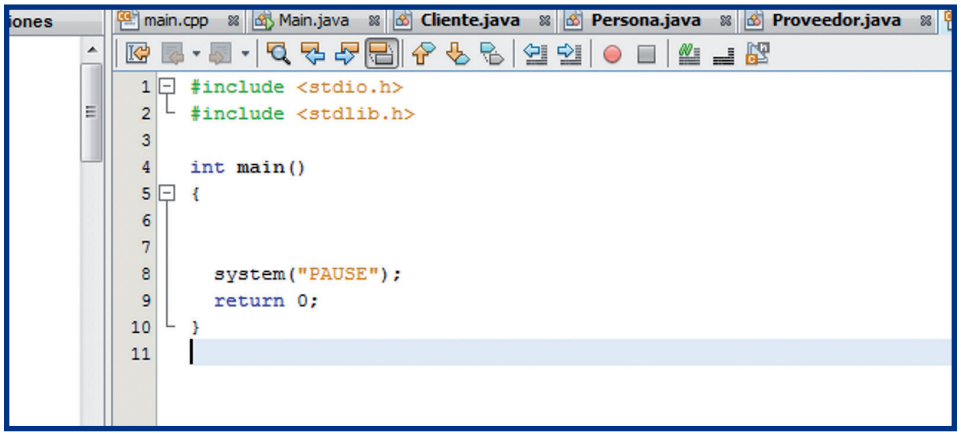


Figura 4.8
En C++, las librerías contienen funciones o clases a partir de includes.

Resumen

Una clase es un tipo de datos que se compone de un conjunto de variables denominadas miembros o propiedades, y de un conjunto de funciones denominadas métodos.

Las propiedades y los métodos se pueden clasificar según su visibilidad en públicos y privados, es decir, accesibles desde fuera de la clase o únicamente desde dentro, respectivamente. De esta forma, se dice que el diseñador encapsula la clase, publicando aquellas propiedades (en ortodoxia ninguna) y métodos que interese, definiendo un comportamiento de la clase hacia el exterior.

Además de esta clasificación, las propiedades y los métodos pueden ser estáticos. Una propiedad estática es una variable definida a nivel de clase, es decir, compartida por todos sus objetos. Un método estático es aquel que puede ser llamado con el formato "nombre de clase.nombre de método" sin necesidad de instanciar ningún objeto.

Una clase puede heredar de otra (superclase) mediante la palabra reservada "extends". La clase heredará las propiedades y métodos de la superclase, excepto los métodos constructores. La clase heredera puede redefinir los métodos de la superclase y puede llamar a los métodos de la superclase con la palabra reservada super.

Cuando aplicamos herencia, toma relevancia el tipo de visibilidad protected, que permite que los miembros sean accesibles por las clases derivadas.

Se denomina empaquetado de clases en Java a la creación de un archivo (paquet) de clases que contiene un conjunto de clases con un objetivo común.

Ejercicios de autocomprobación

Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):

1. Las variables miembro y los métodos definidos en una clase serán compartidos o heredados por todos los objetos que de ella deriven.

2. Un principio de la POO es la ocultación, que indica que para modificar o acceder a una variable propia de una clase es preciso utilizar un paso intermedio llamado instanciación.

3. Cuando hablamos de atributos de una clase, nos referimos a todo dato destinado a ser compartido por todos los objetos que de ella deriven.

4. Una propiedad estática (static) es aquella que es única a nivel de objeto, no de clase.

5. La visibilidad consiste en definir los parámetros que, a su vez, definirán el ocultamiento de los elementos de una clase.

6. Los métodos privados (private) solo pueden ser llamados a nivel interno de la clase. Hacen referencia a funciones propias de un objeto, algo que este no tiene por qué compartir con otras clases u objetos derivados.

7. La herencia facilita la creación de objetos a partir de otros ya definidos a través de la obtención de características (métodos y atributos) comunes.

Completa las siguientes afirmaciones:

8. Los _____ sirven para conocer el valor de un atributo en concreto. Es un método de solo lectura, que se limita a devolver el _____ de un único dato en concreto y, en ningún caso, podrá modificarlo.

9. Los constructores de una clase en Java se diferencian en los _____ con los que se llaman desde la instrucción _____. No devuelven nada y se llaman como la _____.

10. Los tres conceptos básicos bajo los que se rigen las clases son los siguientes: la _____, la _____ y la _____.

Soluciones de los ejercicios de autocomprobación

Unidad 4

1. V
2. F. Un principio de la POO es la ocultación, que indica que para modificar o acceder a una variable propia de una clase es preciso utilizar un paso intermedio llamado mensaje.
3. V
4. F. Una propiedad estática (static) es aquella que es única a nivel de clase, no de objeto.
5. V
6. V
7. V
8. Getters, valor.
9. parámetros, new, clase.
10. encapsulación, ocultación, visibilidad.

ÍNDICE

Desarrollo de aplicaciones multiplataforma

MÓDULO:

Programación

UNIDAD:

Lectura y escritura de la información

CEAC
FP Oficial

MÓDULO:
Programación

UNIDAD:
Lectura y escritura de la información

CEAC
FP Oficial

ÍNDICE

1. Introducción

Cuando se empieza en el mundo de la programación, lo primero que se debe tener claro es que un programa es la resolución de un algoritmo cuyo esquema más general puede ser:

- Entrada de datos.
- Transformación de datos.
- Salida de datos.

Por cuestiones pedagógicas, el aprendizaje de la programación pasa por un primer momento donde la entrada de datos es siempre por el teclado y la salida es en la consola (en la pantalla).

Pero existe un gran número de situaciones donde nos interesa cambiar la entrada y/o la salida de datos por ficheros. Es decir, **leer los datos** desde un fichero del sistema y **escribir los resultados** de nuestro programa en un fichero del sistema.

Pero esto no es todo, pues nos dejaríamos otro gran número de situaciones donde la entrada de datos debe realizarse desde un formulario en una ventana del sistema operativo y, en la salida de nuestro programa, deseamos mostrarla en esa u otra ventana del sistema operativo.

También deberíamos contar con una circunstancia de que la entrada de datos o bien, la salida de estos se realice de forma automática desde o hacia una base de datos, tras crear en nuestro programa una conexión que lo haga posible.

En esta unidad vamos a abordar estos dos tipos de entrada y salida de datos. En primer lugar, nos dedicaremos a estudiar la entrada y salida de datos sobre ficheros. En Java, la comunicación entre nuestro programa y los ficheros se efectúa abriendo un flujo de datos (**stream**) y todo ello se gestiona con las clases InputStream y OutputStream (**Reader y Writer para caracteres**). Con estas clases y, sobre todo, las clases que heredan de ellas.

En segundo lugar, trataremos la entrada y salida gráfica, es decir hacia y desde ventanas, estudiando además los formularios, sus controles y eventos, como por ejemplo, un botón y el evento "al hacer clic". En Java lo haremos mediante las clases contenidas en paquetes de AWT (**Abstract Window Toolkit**), un sistema que permite a nuestros programas Java ser independiente del sistema de gráficos y ventanas del sistema operativo.

2. Objetivos pedagógicos

Los objetivos de aprendizaje que se pretenden alcanzar con esta unidad son:

Comprender los conceptos fundamentales de la programación y el proceso general de resolución de un algoritmo.

Explorar diferentes métodos de entrada y salida de datos en programación.

Adquirir habilidades en la programación gráfica con Java utilizando AWTop-down.

3. Lectura y escritura de la información

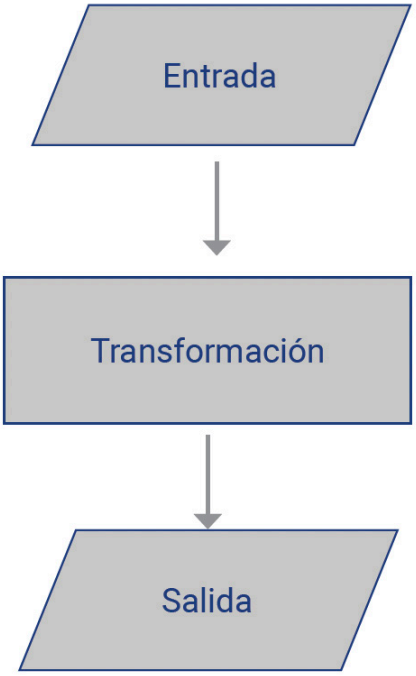
3. 1. Concepto de flujo

En programación, definimos flujo (**stream**) a la conexión existente entre un programa y una fuente o destino de datos. Un flujo define el protocolo usado por los lenguajes de programación para controlar y tratar la entrada (**input**) y salida (**output**) de datos.

Los flujos de entrada (input) controlan los datos que fluyen hacia un programa para ser procesados. Los **flujos de salida (output)** controlan los datos que fluyen desde un programa después de ser procesados. En programación, entendemos por dato aquella parte representativa de información proporcionada u obtenida del programa.

Como ejemplo de entrada de datos podríamos tomar instrucciones del teclado o datos de un archivo, y como ejemplo de salida de datos podríamos tomar la información mostrada a través de la pantalla o un documento enviado a impresora. De este modo se unifica y simplifica la manera en la que el programador puede usar los dispositivos de entrada /salida.

El manejo de flujos en la programación es esencial para interactuar con el entorno, permitiendo la entrada y salida de datos de manera controlada y eficiente. Proporciona flexibilidad, modularidad, capacidad de manipulación de datos y la gestión adecuada de errores, contribuyendo al desarrollo de aplicaciones robustas y funcionales.



Esquema conceptual de entrada y salida de datos en un programa informático.

3.2 Tipos de flujos: flujos de bytes y de caracteres

Existen dos tipos básicos de flujos de datos y ambos están destinados a permitir el uso de ficheros externos a nuestro programa:

- **Flujos de bytes** . El flujo de bytes es un tipo de flujo que se utiliza para la transferencia de datos a nivel de bytes individuales. Este tipo de flujo se utiliza principalmente para leer y escribir datos binarios, como archivos de imágenes, archivos ejecutables, archivos de audio, entre otros.

  En un flujo de bytes, los datos se leen o escriben en su forma más básica, es decir, como una secuencia de bytes sin procesar. Esto significa que no hay ninguna interpretación especial de los datos en términos de caracteres o formato. Los datos se transfieren tal como están almacenados en el archivo o en la memoria.

- **Flujos de caracteres**. El flujo de caracteres es un tipo de flujo que se utiliza para la transferencia de datos en forma de caracteres. Este tipo de flujo se utiliza principalmente para leer y escribir datos de texto, como archivos de texto, datos ingresados por el usuario o datos que se mostrarán en la pantalla.

  En un flujo de caracteres, los datos se leen o escriben en forma de caracteres, utilizando una codificación específica, como UTF-8 o ASCII. Esto permite interpretar los datos como texto legible y realizar operaciones específicas relacionadas con el procesamiento de texto, como búsqueda, manipulación y visualización.

La principal diferencia entre el flujo de bytes y el flujo de caracteres radica en la forma en que se manejan los datos. Mientras que el flujo de bytes se ocupa de los datos a nivel de bytes individuales, el flujo de caracteres se ocupa de los datos a nivel de caracteres.

El u**so adecuado de cada tipo de flujo** depende del tipo de datos que se esté leyendo o escribiendo. Si los datos son binarios y no están destinados a ser interpretados como texto, como archivos multimedia o archivos ejecutables, el flujo de bytes es la opción más adecuada. Proporciona una transferencia directa de los datos sin ninguna interpretación adicional.

Por otro lado, si los datos son texto legible y requieren operaciones específicas relacionadas con el procesamiento de texto, como búsqueda, manipulación o visualización, el flujo de caracteres es la opción más adecuada. Permite una interpretación adecuada de los caracteres según la codificación utilizada y facilita el manejo de datos de texto.

3.3 Flujos predefinidos

En Java, los **flujos predefinidos,** o estándar, definen procesos a nivel interno que permanecen abiertos a lo largo de la ejecución de un programa. Dichos procesos están dispuestos tanto a recibir datos de entrada como a dirigir datos de salida hacia el dispositivo (pantalla, impresora...) o elemento (archivo) designado.

Por defecto, el dispositivo de flujo estándar de entrada es el teclado, y la clase y objeto para referirnos a él es System.in. Asimismo, el flujo estándar de salida por defecto es la pantalla, y para referirse a él se usa la clase y objeto **System.out**. Un tercer objeto de la clase **System**, llamado **System.err**, se encarga de recoger y gestionar los posibles errores generados durante el proceso de flujos de entrada y salida de datos.

Los flujos predefinidos System.in, System.out y System.err se utilizan ampliamente en la programación para realizar operaciones de entrada y salida estándar. Proporcionan una forma sencilla de interactuar con el usuario y mostrar resultados en la consola.

Para utilizar estos flujos, se pueden utilizar métodos específicos proporcionados por el lenguaje de programación. Por ejemplo, en Java, se puede utilizar la clase Scanner para leer datos desde System.in y los métodos println o print para imprimir datos en System.out.

Al utilizar los flujos estándar para entrada y salida, es importante considerar la gestión adecuada de excepciones y el manejo de errores. Por ejemplo, al leer datos desde System.in, se deben manejar posibles excepciones relacionadas con la entrada de datos incorrecta o no válida.

3.4 Clases relativas a flujos

Existen en Java una serie de clases dedicadas a **controlar los flujos de entrada y salida** de datos, y que se conocen con el nombre de clases relativas a flujos. Las clases diseñadas para tales propósitos son cuatro: dos para flujos de entrada y salida de bytes y dos para flujos de entrada y salida de caracteres. De ellas se derivan otras subclases destinadas a realizar funciones más específicas. Veamos cuáles son:

Clases relativas a flujos de bytes

- **InputStream:** La clase InputStream es una clase abstracta que proporciona la base para leer datos de entrada en forma de bytes. Proporciona métodos como read() para leer bytes individuales, read(byte[]) para leer un arreglo de bytes y otros métodos relacionados. Las clases derivadas de InputStream incluyen FileInputStream, ByteArrayInputStream, etc.
- **OutputStream:** La clase OutputStream es una clase abstracta que proporciona la base para escribir datos de salida en forma de bytes. Proporciona métodos como write(int) para escribir un byte individual, write(byte[]) para escribir un arreglo de bytes y otros métodos relacionados. Las clases derivadas de OutputStream incluyen FileOutputStream, ByteArrayOutputStream, etc.

Clases relativas a flujos de caracteres

- **Reader**. Implementa todas las funcionalidades de lectura de flujos de caracteres para que estos puedan ser interpretados y procesados por el programa.
- **Writer**. Implementa todas las funcionalidades de escritura de flujos de caracteres para que estos puedan ser mostrados por pantalla o almacenados en archivos.

Además de las clases mencionadas anteriormente, existen clases específicas que se utilizan para el manejo de flujos en diferentes contextos. Estas clases están diseñadas para adaptarse a escenarios particulares y proporcionan funcionalidades adicionales o especializadas.

Algunos ejemplos de clases específicas para el manejo de flujos son:

- **BufferedInputStream y BufferedOutputStream**: Estas clases proporcionan un búfer interno que mejora el rendimiento al leer y escribir datos en flujos de bytes.
- **BufferedReader y BufferedWriter:** Estas clases proporcionan un búfer interno que mejora el rendimiento al leer y escribir datos en flujos de caracteres.
- **DataInputStream y DataOutputStream:** Estas clases permiten la lectura y escritura de datos primitivos en forma de tipos de datos básicos (int, double, boolean, etc.) en flujos de bytes.

3.5 Utilización de flujos

Para **utilizar flujos** en la programación, es necesario abrirlos y cerrarlos correctamente. La apertura de un flujo implica establecer una conexión con la fuente o destino de datos, mientras que el cierre implica finalizar esa conexión y liberar los recursos asociados al flujo.

- **Apertura de flujos:** La apertura de un flujo implica crear una instancia de la clase correspondiente y establecer la conexión con la fuente o destino de datos. Por ejemplo, al abrir un flujo de lectura desde un archivo, se crea una instancia de la clase FileReader y se especifica el archivo a leer. Del mismo modo, para abrir un flujo de escritura en un archivo, se crea una instancia de la clase FileWriter y se especifica el archivo donde se escribirán los datos.
- **Cierre de flujos:** El cierre de un flujo implica invocar el método adecuado para finalizar la conexión y liberar los recursos asociados. Esto es importante para evitar fugas de recursos y asegurar que los datos se escriban o lean correctamente. Para cerrar un flujo, se utiliza el método close() correspondiente. Por ejemplo, para cerrar un flujo FileReader, se invoca el método close() en la instancia correspondiente.
- **Lectura de datos:** Para leer datos de un flujo, se utilizan métodos como read() (para leer un solo byte o carácter), read(byte[]) o read(char[]) (para leer una secuencia de bytes o caracteres), entre otros. Estos métodos devuelven los datos leídos o un valor especial para indicar el final del flujo.
- **Escritura de datos:** Para escribir datos en un flujo, se utilizan métodos como write(int) (para escribir un solo byte o carácter), write(byte[]) o write(char[]) (para escribir una secuencia de bytes o caracteres), entre otros. Estos métodos toman como argumento los datos a escribir.

Los flujos estándar son gestionados a nivel interno por el sistema: él se encarga de su apertura y de su cierre según sea necesario.

3.6 Entrada desde teclado

La entrada desde el teclado es una forma común de interactuar con un programa y permite que el usuario proporcione datos en tiempo de **ejecución**. Para capturar datos desde el teclado en la mayoría de los lenguajes de programación, se utilizan funciones o métodos específicos que permiten leer la entrada del usuario.

En lenguajes como Java, se puede utilizar la clase Scanner para leer datos desde el teclado. Se crea una instancia de la clase Scanner asociada al flujo de entrada estándar (System.in) y se utilizan los métodos de esta clase, como nextInt(), nextLine(), etc., para leer los datos introducidos por el usuario.

En otros lenguajes como **C++**, se puede utilizar la función cin para leer datos desde el teclado. Se declaran variables correspondientes a los tipos de datos que se desean capturar y se utiliza el operador de extracción (>>) para almacenar los valores introducidos por el usuario en esas variables.

Una vez que se han capturado los datos desde el teclado, es posible realizar operaciones y procesamientos adicionales en función de los requisitos del programa.

Por ejemplo, se pueden realizar **operaciones aritméticas o lógicas** con los datos capturados, realizar comprobaciones de validación o aplicar algoritmos específicos en función de los valores introducidos por el usuario.

También es posible almacenar los datos capturados en variables o estructuras de datos para su posterior uso en el programa.

Es importante **considerar la validación de la entrada del usuario** para asegurarse de que los datos introducidos sean correctos y estén en el formato esperado. Esto puede implicar comprobar si los datos son numéricos, si cumplen ciertas restricciones de longitud o formato, o incluso utilizar expresiones regulares para realizar comprobaciones más avanzadas.

Además, se pueden utilizar bucles para permitir múltiples capturas de datos o para controlar la finalización de la entrada del usuario según ciertas condiciones.

RECUERDA:El flujo es el concepto que define el proceso de entrada y salida de información en un programa informático para unificar el acceso a toda la entradasalida.

3.7. Salida a pantalla

Generalmente, se considera la **pantalla** como el dispositivo principal de salida de datos del sistema o programa.

Los datos enviados hacia la pantalla por el programa se gestionan mediante el objeto System.out perteneciente a la clase OutputStream. Este objeto será el encargado de tratar los flujos de bytes enviados, y de traducirlos a caracteres o resultados interpretables por una persona.

Igual que para los **flujos de entrada**, en lenguaje Java se incluyen una serie de clases designadas para tales efectos; sin embargo, tenemos también la posibilidad de crear nuestras propias clases de control a partir de ellas.

Por defecto, los flujos de datos de salida se muestran por pantalla a través de la consola de nuestro sistema operativo. No obstante, una vez terminado el programa, podremos definir que dichos datos sean mostrados por pantalla representados de un modo más visual (como, por ejemplo, en gráficos) e incluso ofrecer la posibilidad de que sean manipulados por el usuario a través de entradas de teclado.

PARA SABER MÁS: Combinando las funciones que ofrecen la clases relativas a flujos, seremos capaces de crear nuevas funcionalidades adaptadas a las necesidades de nuestros programas.

En la práctica, lo más habitual en **Java** para escribir en la consola es tal y como se muestra en el siguiente ejemplo:

```
23
24      static final int MAX = 10; // Constante
25
26      static int valores[];
27
28      public static void main(String[] args) throws IOException {
29
30
31          int i;
32          valores=new int [MAX];
33          for (i=0 ; i <MAX ; i++){// Introducir 10 valores
34                valores[i]= inputInt();
35
36          }
37          System.out.println("Valores introducidos ");
38          for (i=0; i <MAX ; i++ ){ // Visaulizar 10 valores
39                System.out.println("Posición " + i + " y valor " +  valores[i]  );
40          }
41      }
```

ejercicio1.Ejercicio1

Output - Ejercicio1 (run)

```
10
Valores introducidos
Posición 0 y valor 1
Posición 1 y valor 2
Posición 2 y valor 3
Posición 3 y valor 4
Posición 4 y valor 5
Posición 5 y valor 6
Posición 6 y valor 7
Posición 7 y valor 8
Posición 8 y valor 9
Posición 9 y valor 10
BUILD SUCCESSFUL (total time: 8 seconds)
```

Ejemplo de uso de la consola para verificar una aplicación.

En otros **lenguajes como C++**, se utiliza la función cout para imprimir datos en la pantalla. Se utiliza el operador de inserción (<<) para enviar los datos a la salida estándar. Se pueden imprimir cadenas de texto, variables y otros tipos de datos directamente utilizando el operador de inserción.

Además de imprimir datos en la pantalla, a menudo es necesario formatear y presentar la salida de manera adecuada. Esto implica controlar el diseño, la alineación, la precisión y otros aspectos de cómo se muestra la información al usuario.

En muchos **lenguajes de programación**, se pueden utilizar secuencias de escape o caracteres especiales para formatear la salida. Por ejemplo, se pueden utilizar caracteres como "\n" para realizar saltos de línea, "\t" para insertar tabulaciones o "%d" para representar un número entero en un formato específico.

Algunos lenguajes también proporcionan funciones o métodos para realizar un formateo más avanzado de la salida, como la función printf en C o los métodos de la clase String en Java.

Es importante considerar el tipo de datos que se va a imprimir y el formato deseado para presentarlos de manera legible y comprensible para el usuario. Esto puede incluir la especificación de la cantidad de decimales a mostrar, la alineación de columnas, el uso de caracteres especiales o el formato de fechas y horas, entre otros aspectos.

3.8 Aplicaciones del almacenamiento de información en ficheros

La **filosofía de almacenar información en ficheros externos** a nuestro programa es, a parte de la lógica necesidad de poder guardar unos datos de manera que no sean volátiles y que sean accesibles para futuros usos, la de separar la programación de la información.

De este modo, pueden modificarse los datos de entrada a nuestro programa independientemente de cuál sea su código. Imaginemos que tenemos un programa diseñado para mostrar una galería de fotos. Para indicar al programa cuál es la ubicación de cada uno de los archivos de imagen, deberemos especificar una ruta para cada uno de ellos.

Una primera posibilidad es la de incluir la ruta de cada imagen dentro del mismo código. De este modo, el programa no dejará de ser funcional, pero su usabilidad y mantenimiento perderán eficacia, ya que cualquier modificación de la información, como el cambio del nombre de un archivo, afectará directamente al código de la aplicación.

RECUERDA: Se considera que el teclado y la pantalla son los dispositivos principales de entrada y salida de datos en un programa informático; sin embargo, en este proceso pueden intervenir muchos más dispositivos, como la impresora, una webcam o el ratón.

```
▼<fe:Facturae xmlns:ds="http://www.w3.org/2000/09/xmldsig#" xmlns:fe="http://www.facturae.es/Facturae/2009/v3.2/Facturae">
  ▼<FileHeader>
     <SchemaVersion>3.2</SchemaVersion>
     <Modality>I</Modality>
     <InvoiceIssuerType>EM</InvoiceIssuerType>
    ▼<Batch>
       <BatchIdentifier>0000000000B18</BatchIdentifier>
       <InvoicesCount>1</InvoicesCount>
      ▼<TotalInvoicesAmount>
         <TotalAmount>63.13</TotalAmount>
       </TotalInvoicesAmount>
      ▼<TotalOutstandingAmount>
         <TotalAmount>63.13</TotalAmount>
       </TotalOutstandingAmount>
      ▼<TotalExecutableAmount>
         <TotalAmount>63.13</TotalAmount>
       </TotalExecutableAmount>
       <InvoiceCurrencyCode>EUR</InvoiceCurrencyCode>
     </Batch>
   </FileHeader>
  ▼<Parties>
    ▼<SellerParty>
      ▼<TaxIdentification>
         <PersonTypeCode>J</PersonTypeCode>
         <ResidenceTypeCode>R</ResidenceTypeCode>
         <TaxIdentificationNumber>A82735122</TaxIdentificationNumber>
       </TaxIdentification>
      ▼<LegalEntity>
         <CorporateName>Company Comp SA</CorporateName>
         <TradeName>Comp</TradeName>
        ▼<RegistrationData>
```

Ejemplo de un archivo XML. Formato de factura electrónica.

3.9 Ficheros de datos. Registros

3.9.1 Registros del sistema

Un **fichero de datos es una entidad lógica** que se utiliza para almacenar y organizar información de manera persistente en un sistema de almacenamiento, como un disco duro o una memoria externa. Es una estructura de datos que permite almacenar datos de diferentes tipos, como texto, números, fechas, imágenes, entre otros, de manera estructurada y accesible.

Un fichero de datos puede contener uno o varios registros, que son unidades lógicas de información dentro del fichero. Cada registro representa una entidad o una colección de datos relacionados que se almacenan como un conjunto. Por ejemplo, en una base de datos de empleados, cada registro puede representar a un empleado específico y contener información como nombre, edad, dirección y salario.

Las versiones de **Java** a partir de la versión 1.4 añaden una funcionalidad que permite controlar, dar formato y publicar mensajes a través de los llamados registros (log). Los registros pertenecen a un paquete de clases llamado java.util.logging. Los registros pueden ser usados para lanzar mensajes de información, estados de los datos o incluso errores ocurridos durante la ejecución del programa. Estos procesos pueden beneficiar a los usuarios de la aplicación sea cual sea su perfil. El proceso de utilización de los registros de Java es muy simple, sólo hay que seguir los siguientes pasos:

- Incluir las sentencias de importación para el paquete de clases logging.
- Creación de una referencia a la clase Logger.
- Indicar el mensaje que desea mostrarse y su tipología.

Ejemplo de implementación de la clase Logger en código Java

```
import java.io.*;
import java.util.logging.*;
public class QuintLog{
        public static void main(String[] args) {
        try{
        FileHandler hand = new FileHandler("vk.log");
        Logger log = Logger.getLogger("log_file");
        log.addHandler(hand);
        log.warning("Doing carefully!");
        log.info("Doing something...");
        log.warning("Doing strictily");
        System.out.println(log.getName());
        }
        catch(IOException e){}
        }
}
```

3.9.2 Estructura de registros en ficheros

La **estructura de registros en un fichero** define cómo se organizan y almacenan los datos dentro del fichero. Cada registro generalmente consiste en campos individuales que representan atributos o características específicas de la entidad que se está representando. Estos campos pueden tener diferentes tipos de datos, como texto, números o fechas.

Los registros pueden tener una estructura fija o variable. En una estructura fija, cada registro tiene la misma cantidad de campos y cada campo tiene un tamaño predefinido. Esto facilita el acceso a los datos, ya que se puede acceder a un campo específico mediante su posición o desplazamiento dentro del registro.

En una **estructura variable**, los registros pueden tener diferentes tamaños y el número y tipo de campos pueden variar entre registros. Esto brinda flexibilidad para adaptarse a diferentes situaciones, pero también puede complicar el acceso a los datos, ya que se requiere información adicional para determinar la ubicación y el tamaño de cada campo dentro del registro.

Además de los campos individuales, los registros también pueden incluir información adicional, como un identificador único o una clave primaria, que se utiliza para identificar y buscar registros específicos dentro del fichero.

La estructura de registros en un fichero puede variar según el lenguaje de programación, el sistema de gestión de bases de datos o la aplicación específica que se esté utilizando. La elección de la estructura adecuada depende de los requisitos de la aplicación y la eficiencia en el acceso y manipulación de los datos.

Suponiendo que se desee guardar la información de la clase Pieza, con los campos, código, nombre y precio, las dos opciones serían:

Tamaño fijo

Ejemplo de registros de tamaño fijo

```
000001ConectorRJ45 0001.45
000002CableUTP6e 0009.27
009871ImpresoraRh99 0123.67
```

Tamaño variable y separadores

```
1#ConectorRJ45#.45
2#CableUTP6e#9.27
9871#ImpresoraRh99#123.67
```

En este caso se usa el símbolo # como separador de campos y el salto de línea como separador de registros.

RECUERDA: El almacenado de información en archivos externos de nuestro programa nos proporcionará un mayor control de nuestras aplicaciones al dejar de depender únicamente del código fuente.

3.10 Apertura y cierre de ficheros. Modos de acceso

Igual que en los procesos designados para **controlar la apertura,** proceso previo necesario para cualquier acceso a un fichero, y cierre de flujos de información, existen también unos protocolos que rigen la apertura y cierre de los ficheros externos usados en nuestro programa.

Abrir un fichero significa **establecer una conexión** entre el programa y el fichero para poder acceder a su contenido, mientras que cerrar un fichero implica finalizar esa conexión y liberar los recursos utilizados.

Para abrir un fichero, se utiliza una función o método proporcionado por el lenguaje de programación, donde se especifica el nombre del fichero y el modo de acceso deseado. Los modos de acceso pueden ser de lectura, escritura o actualización, y determinan los permisos y operaciones que se permiten sobre el fichero.

Una vez que se ha terminado de utilizar un fichero, es importante cerrarlo adecuadamente para liberar los recursos asociados y garantizar la integridad de los datos. Esto se realiza mediante una función o método de cierre proporcionado por el lenguaje de programación, que garantiza que todos los datos pendientes se escriban en el fichero y se liberen los recursos utilizados.

A través del paquete de **clases FileReader**, Java proporciona toda una serie de funciones destinadas a la apertura, lectura y cierre de ficheros. Por medio de una ruta definida por un archivo alojado en nuestro disco duro, o por una URL de Internet, se indica a la clase **FileReader** cuál es el origen de los datos.

Otras clases y funciones como **BufferedReader** o readLine se encargarán de recoger en forma de datos la información contenida en los ficheros y almacenarla en variables de nuestro programa.

Una vez terminado el proceso, finalizaremos cerrando el fichero y liberando de la memoria todos los recursos que habían sido utilizados para su tratamiento.

Es buena práctica abrir un fichero justo antes de su uso y cerrarlo tan pronto como ya no sea necesario. Esto garantiza un manejo eficiente de los recursos del sistema y evita problemas como la corrupción de datos o bloqueos de acceso.

3.11 Escritura y lectura de información en ficheros

La escritura y lectura de información en ficheros son operaciones fundamentales en el manejo de archivos. Estos procesos permiten **almacenar datos en un fichero** y recuperarlos posteriormente según sea necesario.

Para escribir información en un fichero, se utiliza un flujo de salida (OutputStream o Writer) que proporciona el lenguaje de programación. Se puede escribir datos en el fichero de forma secuencial o estructurada, dependiendo de los requisitos del programa. La escritura secuencial implica escribir los datos uno tras otro en el fichero, mientras que la escritura estructurada implica escribir datos en registros o estructuras de datos específicas.

Para leer información desde un fichero, se utiliza un flujo de entrada (InputStream o Reader). Se pueden leer los datos del fichero secuencialmente o de forma estructurada, de acuerdo con la forma en que se escribieron. La **lectura secuencial** implica leer los datos en el mismo orden en que se escribieron, mientras que la lectura estructurada implica leer datos de registros o estructuras específicas.

Ejemplo de código Java con implementación de lectura y escritura de un archivo

```
package ejemploLecEscri;
import java.io.*;
public class CopyFile{
        public static void main(String[] args) {
              File origen = new File(args[0]);
              File destino = new File(args[1]);
              FileInputStream fileInput = null;
              FileInputStream fileOutput = null;
              try
              {
                 fileInput = new FileInputStream(origen);
                 fileOutput = new FileOutputStream (destino);
                 byte [] array = new byte[4096];
                 while (true)
                 {
                     int n = fileInput.read(buffer);
                     if(n>0){
                     fileOutput.write(buffer,0,n);
                     } else {
                     System.out.println(args[0] + "copiado a: " + args[1]);
                     break;}
                  }
              }
              catch (Exception e)
              {
                System.out.println(e);
              } finally {
                  try {
                     if(fileInput!=null){
                       fileInput.close();}
                     if(fileOutput!=null){
                       fileOutput.close();}
                  } catch (IOException e1) {
                    System.out.println("Mensaje error cierre fichero: " + e1.getMessage());}
              }
}
```

Este proceso nos permitirá tener información almacenada externamente para poder acceder a ella en puntos determinados del programa. Además, nos permitirán separar la programación de la información, pudiendo modificar así los datos de entrada independientemente de cuál sea nuestro código.

La **tipología de archivos** que podrá ser utilizada y manipulada por nuestros programas es muy diversa, puede tratarse de simples documentos de texto con extensión .txt, o archivos de información estructurada como un .xml, archivos de imagen, o incluso archivos ejecutables.

Cuando manejamos archivos de texto, se utilizan FileReader y FileWriter, junto con otras clases tipo Buffered para leer y escribir datos (en este caso texto). En el caso de otro tipo de ficheros, donde manejamos bytes, se utilizan las clases InputStream y OutputStream, además de sus derivadas.

Al escribir y leer información en ficheros, es importante considerar el formato de los datos. El formato de datos se refiere a la estructura y codificación utilizada para almacenar los datos en el fichero.

Algunos formatos comunes incluyen:

- **Formato de texto plano:** Los datos se almacenan en el fichero como texto sin formato, utilizando caracteres legibles por humanos. Este formato es legible y puede ser editado manualmente, pero puede ocupar más espacio en el disco.
- **Formato binario:** Los datos se almacenan en el fichero en un formato binario, utilizando una representación más eficiente en términos de espacio y velocidad de lectura/escritura. Este formato es más compacto pero, no es legible por humanos.
- **Formato CSV (Comma-Separated Values):** Los datos se almacenan en el fichero como texto separado por comas, donde cada campo de datos se representa como un valor separado por comas. Este formato es ampliamente utilizado para el intercambio de datos estructurados.
- **Formato XML (eXtensible Markup Language):** Los datos se almacenan en el fichero utilizando etiquetas y atributos, siguiendo una estructura jerárquica. El formato XML es ampliamente utilizado para el intercambio de datos y permite la representación de datos estructurados y metadatos.
- **Formato JSON (JavaScript Object Notation):** Los datos se almacenan en el fichero en formato de texto utilizando una sintaxis similar a la de los objetos en JavaScript. El formato JSON es ampliamente utilizado en aplicaciones web y servicios web.

3.12 Almacenamiento de objetos en ficheros. Persistencia. Serialización

La **serialización** es el proceso a través del cual pueden almacenarse objetos directamente en una secuencia de bytes para poder almacenarlo en un fichero. Esta secuencia de bytes puede ser posteriormente utilizada para recrear el objeto en la memoria. La serialización es útil cuando se necesita persistir objetos en disco o transmitirlos a través de una red.

Para serializar un objeto, se utiliza el mecanismo de serialización proporcionado por el lenguaje de programación. El objeto a serializar debe implementar la interfaz Serializable, que indica que el objeto puede ser convertido en una secuencia de bytes. A través de la serialización, se guardan todos los datos y el estado del objeto en el fichero.

Se trata de un proceso basado en el concepto de la **persistencia** que se refiere a la capacidad de almacenar objetos en ficheros de forma duradera, de modo que puedan ser recuperados y utilizados en futuras ejecuciones del programa.

A lo largo del proceso de **serialización**, los datos del objeto se escriben en un flujo de salida (OutputStream) y se guardan en el fichero. Esto incluye la escritura de los campos de datos y los valores de las variables del objeto. Al finalizar la serialización, el objeto se ha convertido en una secuencia de bytes almacenada en el fichero.

Para recuperar un objeto serializado desde un fichero, se realiza el proceso inverso, llamado deserialización. La deserialización implica leer la secuencia de bytes almacenada en el fichero y recrear el objeto original en la memoria.

Durante la deserialización, se utiliza un flujo de entrada (**InputStream**) para leer los datos del fichero. Los datos se transforman nuevamente en un objeto, utilizando la información almacenada en la secuencia de bytes. Es importante destacar que el objeto recuperado será una copia exacta del objeto original, incluyendo su estado y datos.

Mediante los métodos **ObjectOutputStream y ObjectInputStream**, podremos realizar operaciones complejas con nuestros objetos como, por ejemplo, enviarlos a través de la red en formato de bytes y reconstruirlos de nuevo una vez lleguen a su destinatario.

```java
/**
 * Escribe a archivo el objeto Empleado
 *
 * @param archivo Nombre de archivo
 * @throws IOException Excepción de entrada/salida
 */
public void serializar (String archivo) throws IOException {
    ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream(archivo))
    salida.writeObject(this);
}
```

Código Java con ejemplo del proceso de serialización y persistencia.

3.13 Utilización de los sistemas de ficheros

Un sistema de **ficheros** es la representación de todos los elementos incluidos en nuestro ordenador, ya sean unidades (en algunos sistemas operativos), carpetas, archivos, programas o dispositivos externos, como impresoras o escáneres.

La utilización de los sistemas de ficheros en la programación implica interactuar con el sistema operativo para realizar operaciones de lectura, escritura y manipulación de ficheros y directorios. Cada sistema operativo proporciona una interfaz de programación (API) específica para acceder y manipular los recursos del sistema de archivos.

Para interactuar con el sistema de archivos, se utilizan funciones o métodos proporcionados por el **lenguaje de programación.** Estas funciones permiten realizar operaciones como crear, abrir, leer, escribir, mover, copiar y eliminar ficheros y directorios.

Java incluye una serie de **funcionalidades** para poder interactuar con todos ellos. Igual que podemos detectar una entrada de teclado y registrar una acción a partir de ello, podremos también acceder a métodos que activen funciones de impresión, modificación de archivos o ejecución de otros programas complementarios de apoyo.

El **sistema de archivos** proporciona también al lenguaje la organización de nuestros archivos y directorios en el disco duro, de este modo tendremos la posibilidad de poder seleccionar un archivo concreto para que sea procesado en tiempo de ejecución del programa. Así se podrá escoger un archivo de una ruta, saber si es un archivo o una carpeta, el tipo de archivo, etc.

Ejemplo de sistema de archivos.

PARA SABER MÁS: Los ficheros de registros son una herramienta muy valiosa en cuanto a diseño de aplicaciones se refiere. Suponen el complemento perfecto, ya sea para otros programadores que intervengan en su desarrollo como para los usuarios finales.

3.14 Creación y eliminación de ficheros y directorios

Las **operaciones básicas** de utilización de los sistemas de ficheros incluyen la creación, eliminación y manipulación de ficheros y directorios. Para esto, podemos utilizar el paquete de clases Java IO. Java IO es una API (Application Programming Interface) que proporciona funcionalidades para el manejo de entrada y salida de datos en Java.

Esta **API** está diseñada para trabajar con diferentes tipos de flujos de datos, como flujos de bytes y flujos de caracteres, y ofrece diversas clases y métodos para realizar operaciones de lectura y escritura en archivos, dispositivos y otros canales de comunicación. La clase para llevar a cabo los procesos de creación, eliminación y modificación de los ficheros se llama 'File'.

El método que deberemos utilizar es **createNew-File**, incluido dentro de las funcionalidades de dicha clase. El proceso es bien simple, tan sólo deberemos incluir dos parámetros: el primero indicará la ruta física donde se quiera crear el archivo; el segundo indicará el nombre que recibirá el archivo creado.

El único factor a tener en cuenta es que la r**uta definida** para la creación del archivo deberá existir previamente en nuestro sistema. Hay que señalar que la doble barra usada en la sentencia es la que le indicará a nuestro código la jerarquía o separación de los directorios en la ruta indicada. Una vez creado el fichero, bastará con invocar el método createNewFile, definiendo una estructura de excepción (IOException), que se encargará de recoger los posibles errores que aparezcan a lo largo del proceso.

```
...
import java.io.File;
...
File f = new File("un_path/un_fichero.txt");

if (f.canRead())
   // El fichero existe y se puede leer.

if (f.canWrite())
   // El fichero existe y se puede escribir en él

if (f.canExecute())
   // El fichero existe y se puede ejecutar
```

Código Java que además de crear identifica los permisos.

Como se ha explicado, en **Java**, la gestión de un fichero o de un directorio se efectúa mediante la clase File. Además del método explicado, otra manera de trabajar es con los constructores que podemos ver en los ejemplos siguientes:

```
File f1 = new File("c:\\windows\\notepad.exe"); // La barra '\' se escribe '\\'

File f2 = new File("c:\\windows"); // Un directorio
```

La propiedad estática **File.separator** nos proporciona el carácter separador del sistema de archivos donde se está ejecutando nuestro programa. En el ejemplo anterior, si queremos que nuestro programa funcione en Linux y en Windows, deberíamos utilizar esta propiedad en cualquier referencia a una ruta del sistema de ficheros. Por ejemplo:

```
File f1 = new File("directorio"+File.separator+"directorio"+ …
```

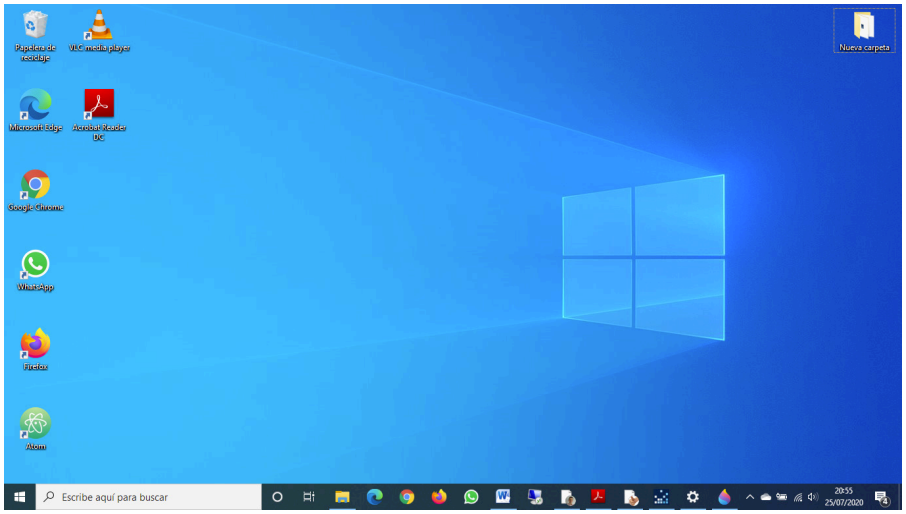
Otra utilidad que nos ofrece Java, es la ventana de diálogo del sistema de ficheros para seleccionar un fichero o directorio concretos. La gestión de estas ventanas de diálogo en Java se efectúa mediante la clase FileDialog el paquete **java.awt.**

Una vez se ha creado el objeto File con éxito, ya sea un directorio o un fichero, el método "delete" permite la eliminación de dicho directorio o fichero del disco. El siguiente ejemplo elimina el fichero **"a.txt"** del disco, controlando si se ha efectuado con éxito. En el caso del directorio, se procedería igual:

```
File fic = new File("a.txt");
if (fic.delete())
   System.out.println("El fichero ha sido borrado satisfactoriamente");
else
   System.out.println("El fichero no puede ser borrado por algún motivo");
```

3.15 Creación de interfaces gráficos de usuario utilizando asistentes y herramientas del entorno integrado

La **interfaz gráfica de usuario** (en inglés Graphical User Interface, GUI), a juega un papel fundamental en el desarrollo de aplicaciones, ya que es la parte visible e interactiva con la cual los usuarios interactúan. Es la que permite presentar la información de manera visualmente atractiva y organizada, facilitando la comprensión y navegación por parte de los usuarios.



Escritorio del sistema operativo Windows como ejemplo de interfaz gráfica de usuario.

La **filosofía de una GUI** es facilitar la interacción del usuario con el ordenador a través de acciones realizadas mediante la manipulación directa de los elementos en pantalla. Una GUI surge de la evolución de la consola de los primeros sistemas operativos, que representa la pieza fundamental de todo entorno gráfico. Como ejemplo de GUI podemos citar el escritorio del sistema operativo Windows. En general, una GUI es el entorno visual que se muestra cuando ejecutamos un programa informático.

En Java hay **diversas utilidades**, paquetes de clases o herramientas, que nos permiten implementar aplicaciones con entorno gráfico. Estas utilidades nos ofrecen la gran ventaja con respecto a otros lenguajes de poder hacer nuestro programa independiente del GUI del sistema operativo.

Entre otras, podemos utilizar la herramienta **AWT (Abstract Window Toolkit) y Swing,** aunque por cuestiones pedagógicas, para aprender a programar es conveniente utilizar AWT.

3.16 Interfaces

Las **interfaces en la programación son abstracciones** que definen un conjunto de métodos que una clase puede implementar. Sirven como contratos o especificaciones de comportamiento, estableciendo qué métodos deben estar presentes en una clase que implemente esa interfaz. Las interfaces permiten la creación de un código modular y flexible, ya que una clase puede implementar múltiples interfaces, lo que facilita la reutilización de código y promueve la interoperabilidad.

Para diseñar y crear interfaces gráficas, existen diversas herramientas y bibliotecas disponibles que facilitan el proceso de desarrollo. Algunas de las herramientas más populares son:

- **IDEs (Entornos de Desarrollo Integrados):** IDEs como Eclipse, IntelliJ IDEA y NetBeans proporcionan herramientas visuales para diseñar interfaces gráficas. Estas herramientas permiten arrastrar y soltar componentes visuales, establecer propiedades y agregar comportamiento a los elementos de la interfaz.
- **Frameworks y bibliotecas GUI:** Hay numerosos frameworks y bibliotecas disponibles que facilitan la creación de interfaces gráficas en Java, como Swing, JavaFX y AWT (Figura 5.10). Estos frameworks proporcionan un conjunto de componentes visuales y herramientas para diseñar interfaces gráficas de manera programática.
- **Herramientas de diseño gráfico:** Programas de diseño gráfico como Adobe Photoshop, Sketch y Figma pueden ser utilizados para crear y editar gráficos, iconos y elementos visuales que se utilizarán en las interfaces gráficas.

Como se ha mencionado anteriormente, Java AWT (Abstract Window Toolkit) es una biblioteca de Java que proporciona componentes gráficos para crear interfaces de usuario en aplicaciones de escritorio.

Aquí, el primer concepto para la **interfaz gráfica** es la separación entre contenedor y componente. La ventana es un contenedor (Frame) y un botón es un componente.

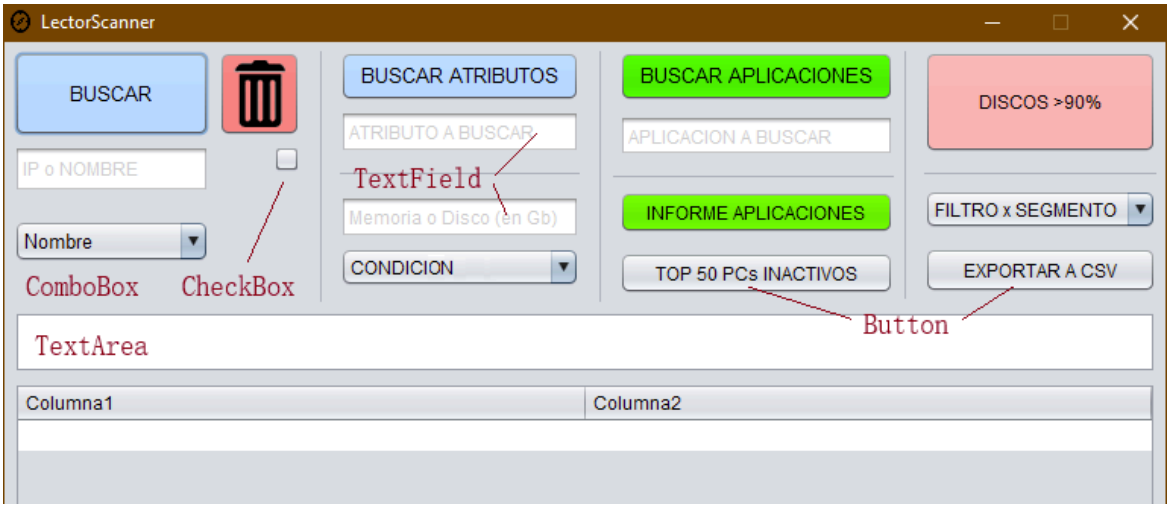


Imagen de una ventana de AWT con una muestra de algunos componentes de este entorno.

Los contenedores de **AWT**:

- Frame (classe java.awt.Frame)
- Panel (classe java.awt.Panel)

Como hemos visto, Frame es una ventana y Panel es un contenedor que podemos colocar dentro de una ventana con el simple propósito de agrupar componentes dentro de él.

Para organizar los contenidos (**contenedores y componentes**) dentro de una ventana se utilizan los layout managers. Pensemos que, a diferencia de otros entornos, no podemos ubicar libremente un componente dentro de una ventana, pero por el contrario, nuestro programa gráfico funcionará en cualquier sistema operativo.

Los principales **layout managers de AWT**:

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.GridLayout

Un **BorderLayout** divide el contenedor en cinco áreas, norte, este, oeste, sur y centro. En un FlowLayout, los componentes se van añadiendo a la derecha del anterior y, si no cabe en la anchura de la ventana, entonces se mostrará en la línea siguiente.

Un **GridLayout** corresponde a una tabla con "n" filas por "m" columnas. Cada componente que se añade se mostrará en la siguiente columna, empezando nueva fila cuando aquellas se hayan acabado.

Los **componentes de AWT**, básicamente controles, son:

- Etiqueta (classe java.awt.Label)
- Botón (classe java.awt.Button)
- Radiobutton / Checkbox (classe java.awt.Checkbox), agrupados o no en un checkboxgroup (classe java.awt.CheckboxGroup)
- Lista (classe java.awt.List)
- Desplegable para seleccionar (classe java.awt.Choice)
- Campos de texto (classe java.awt.TextField)
- Áreas de texto (classe java.awt.TextArea)

3.17 Concepto de evento

En el contexto de las **interfaces gráficas**, un evento es una acción o suceso que ocurre en un componente de la interfaz, como un clic de ratón, pulsación de tecla o cambio de estado. Los eventos son fundamentales en la programación de interfaces gráficas, ya que permiten la interacción del usuario con la aplicación y desencadenan respuestas o acciones por parte del programa.

Los eventos proporcionan una forma de capturar las acciones del usuario y responder a ellas de manera programática. Cuando ocurre un evento, se genera una notificación que es enviada al componente correspondiente, y este a su vez puede ejecutar un código específico para manejar dicho evento. Esto permite crear aplicaciones interactivas y responsivas, donde las acciones del usuario tienen un impacto inmediato en la ejecución del programa.

En Java, la clase dedicada al control de los eventos está incluida dentro del paquete java.awt.Event. Este **paquete proporciona clases e interfaces** para el manejo de eventos en Java. Este paquete es parte de la biblioteca AWT (Abstract Window Toolkit) y contiene diversas clases y subpaquetes que se utilizan para capturar y procesar eventos generados por los componentes de la interfaz gráfica.

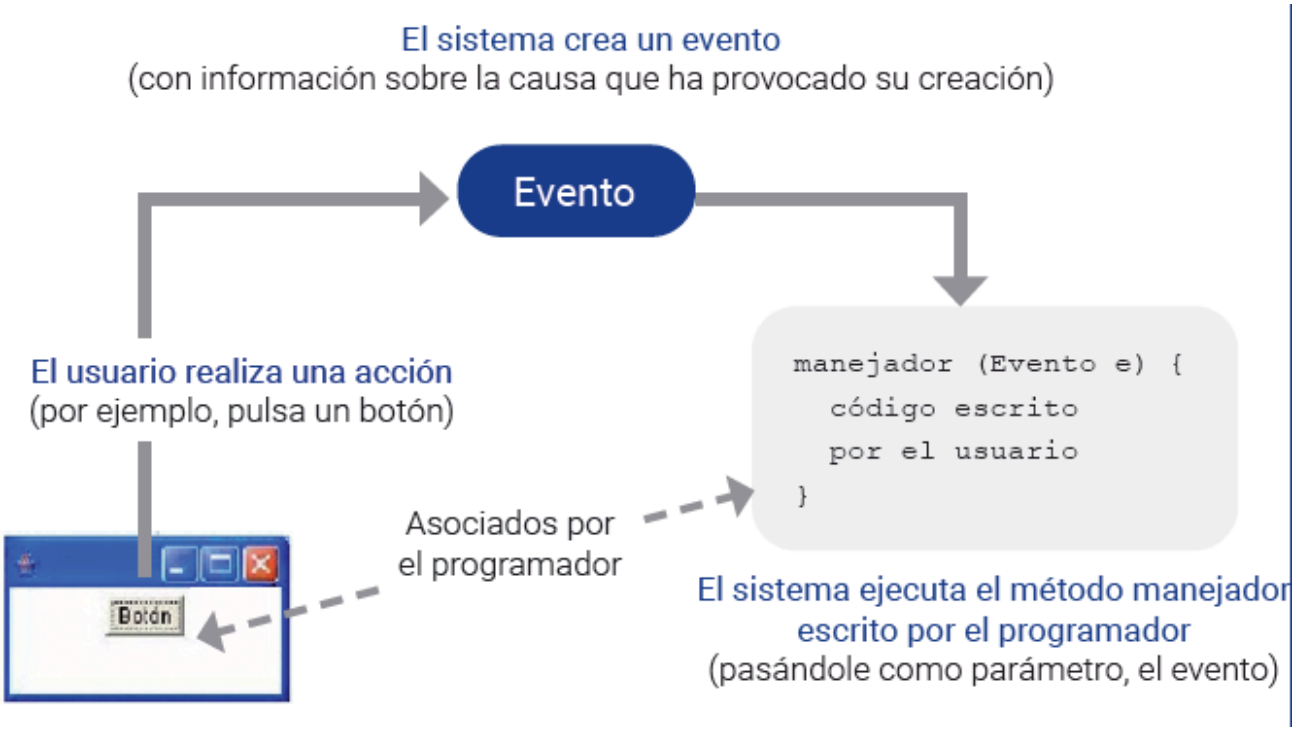
Existen diferentes tipos de eventos que pueden ocurrir en una interfaz gráfica, y cada uno de ellos tiene su propio manejo y respuesta. Algunos ejemplos de tipos de eventos comunes son:

- **Eventos de ratón:** Estos eventos se generan cuando se interactúa con el ratón, como hacer clic, mover el cursor o arrastrar elementos. Algunos ejemplos de eventos de ratón son MouseEvento clic (MouseListener), MouseEvento movimiento (MouseMotionListener) y MouseEvento de arrastre (MouseDragListener).
- **Eventos de teclado:** Estos eventos se generan cuando se presionan o sueltan teclas del teclado. Algunos ejemplos de eventos de teclado son KeyEvento presionar una tecla (KeyListener) y KeyEvento soltar una tecla (KeyListener).

- **Eventos de acción**: Estos eventos se generan cuando se realiza una acción específica en un componente, como hacer clic en un botón o seleccionar un elemento de una lista. Algunos ejemplos de eventos de acción son ActionEvento clic en un botón (ActionListener) y ItemEvento selección de un elemento en una lista (ItemListener).

Para manejar estos eventos, se utilizan interfaces y clases proporcionadas por la biblioteca de Java, como MouseListener, KeyListener y ActionListener. Estas interfaces contienen métodos que se implementan para definir la lógica que se ejecutará cuando ocurra un evento específico. Por ejemplo, al implementar el método ActionPerformed() de la interfaz ActionListener, se puede definir el código que se ejecutará cuando se haga clic en un botón.

Esquema del proceso de escucha y activación de función de un controlador de eventos.



3.18 Creación de controladores de eventos

Un **controlador de eventos** es un método o función designado a lanzar una acción determinada cuando se registra un evento a lo largo de la ejecución de un programa. Los **controladores de eventos** se aplican directamente sobre los objetos definidos en nuestro código. Al aplicar un controlador sobre un objeto, éste recibe una función de escucha para dicho evento determinado.

Por este motivo los controladores de eventos reciben el nombre de **listener**. Por ejemplo, si queremos detectar un clic de ratón en nuestra aplicación, deberemos recurrir a los eventos de ratón (MouseEvent) definidos en las clases del lenguaje.

La implementación de un controlador de eventos implica seguir estos pasos:

- **Identificar el componente y el evento:** Determine qué componente generará el evento y qué tipo de evento se espera capturar. Por ejemplo, si desea capturar un clic de botón, identifique el botón relevante y el evento ActionEvent.
- **Implementar la interfaz de controlador de eventos:** Dependiendo del tipo de evento, implemente la interfaz apropiada. Por ejemplo, para capturar eventos de clic de botón, implemente la interfaz ActionListener.
- **Sobrescribir el método de manejo de eventos:** En la clase que implementa la interfaz de controlador de eventos, sobrescriba el método correspondiente al evento que desea capturar. Por ejemplo, para eventos de clic de botón, sobrescriba el método actionPerformed(ActionEvente).
- **Implementar la lógica del evento:** Dentro del método de manejo de eventos, defina la lógica que se ejecutará cuando ocurra el evento. Por ejemplo, puede realizar operaciones, cambiar el estado de otros componentes o actualizar la interfaz gráfica.
- **Asociar el controlador de eventos al componente:** Finalmente, asocie el controlador de eventos al componente correspondiente utilizando métodos como addActionListener() o addMouseListener() según corresponda.

3.19 Generación de programas en entorno gráfico

La función de los **entornos gráficos de usuario o GUI** es ofrecer un abanico de herramientas para los programadores destinadas a diseñar aplicaciones de un modo visual e intuitivo.

La generación de programas a partir de entornos gráficos nos permite programar una aplicación al tiempo que evaluamos visualmente como será la experiencia final del usuario al que irá destinada.

Gracias al entorno visual que facilitan las GUI podremos crear fácilmente una aplicación con sólo arrastrar botones y otros componentes de las barras de opciones del entorno. No obstante, muchas veces el programador lo hace sin recurrir a estos entornos visuales, pues copia código que ya tiene realizado.

PARA SABER MÁS: El dominio de las clases de los controladores de eventos es uno de los pilares de la programación de aplicaciones informáticas, ya que nos permitirá tener un control preciso de todas las acciones que tengan lugar en un programa.

El esquema general para realizar una aplicación en **Java AWT** es:

- **Crear un proyecto Java normal**, como se ha hecho hasta ahora, por ejemplo, con NetBeans. No crearemos la clase principal Main.java.
- **Añadir una nueva clase**, por ejemplo "Ventana" pero tipo Otros / de interfaz gráfico o AWT GUI y seleccionamos Formulario Frame. Hagamos que se cree también el paquete con el nombre que deseemos.
- Ya podemos **ejecutar el programa**. Si nos pide la clase principal, le decimos el nombre que le hemos dado a nuestro Formulario Frame. Net-Beans nos ha añadido el código necesario.
- En este momento, podríamos **añadir los componentes** que nos hicieran falta, y los eventos sobre ellos. Podríamos cambiar el layout, añadir otros contenedores como paneles e incluso otras ventanas. En la parte práctica, haremos algún ejemplo de ello.

El código generado por **NetBeans** es el siguiente:

```
/*
    * To change this template, choose Tools | Templates
    * and open the template in the editor.
    */
/*
    * Ventana.java
    *
    * Created xxx
    */
package as;
/**
    *
    * @author yyy
    */
public class Ventana extends java.awt.Frame {
    /** Creates new form Ventana */
    public Ventana() {
        initComponents();
    }
    /** This method is called from within the constructor to
    * initialize the form.
```

```
        * WARNING: Do NOT modify this code. The content of this
method is
    * always regenerated by the Form Editor.
    */
        // <editor-fold defaultstate="collapsed" desc="Generated
Code">
    private void initComponents() {
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent
evt) {
                exitForm(evt);
            }
        });
        pack();
    }// </editor-fold>
    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {
        System.exit(0);
    }
    /**
    * @param args the command line arguments
    */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new Ventana().setVisible(true);
            }
        });
    }
    // Variables declaration - do not modify
    // End of variables declaration
}
```

Podemos observar lo siguiente:

- Nuestra clase principal Ventana hereda de Frame.
- Se ha creado un método main, por donde entrará a ejecutarse nuestra aplicación. El código generado (Runnable ...) es para que la ventana se ejecute en un proceso diferente (Thread).
- El constructor se ejecuta cuando creamos el objeto anterior y, por lo tanto, se llama a initComponents().
- El método initComponents añade un listener (el controlador de eventos, el "escuchador") al contenedor Ventana y caza el evento windowClo sing que se produce cuando cerramos la ventana. Al cazar el evento, llamamos a exitForm.
- El método exitForm llama System.exit(0); con lo que el programa finaliza.

Pues esto que nos ha creado **NetBeans** contiene todos los aspectos a considerar en una aplicación gráfica. Se crea un objeto de tipo contenedor, se visualiza, se le añade un listener y se cazan eventos y además se ejecutan métodos cuando se ha cazado el evento. Y esto se puede ir repitiendo con otros contenedores y componentes. La estructura es la misma.

4. Resumen

El concepto de flujo es una referencia amplia del proceso por el que pasa toda la información que interviene en nuestro programa, desde un punto de entrada hasta un punto de salida. Este proceso gestiona el estado de dicha información en cada uno de los puntos de nuestro programa.

En todos los lenguajes de programación existe un control de dicho proceso; sin embargo, sólo en algunos de ellos, como Java, se incluye una serie de clases destinadas a la gestión de los datos por parte del programador.

Operar con ficheros no es una tarea sencilla de controlar; sin embargo, es de gran utilidad en cuanto a tratamiento de la información se refiere. El uso de archivos de apoyo en nuestros programas, tanto para la entrada como para la salida de datos, es una herramienta indispensable que debe dominar todo programador, sea cual sea el lenguaje en el que se especialice.

El uso de ficheros nos ofrece unas ventajas inestimables tanto para organizar la información entrante en nuestro programa como para gestionar la salida de datos o el reporte de errores con respecto a los usuarios finales de la aplicación.

Gracias a los controladores de eventos podremos controlar de forma precisa qué acciones se desencadenan en puntos concretos de nuestro código, obteniendo así un acceso directo a posibles errores que puedan surgir en él.

Las interfaces gráficas de usuario (**GUI**) representan una herramienta muy versátil a la hora de diseñar aplicaciones, ya que, gracias al entorno visual que ofrecen, permiten programar de un modo sólido y, al mismo tiempo, evaluar la experiencia final y la usabilidad de la aplicación diseñada.

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

MÓDULO:
Programación

UNIDAD:
Aplicación de las estructuras de almacenamiento

CEAC
FP Oficial

MÓDULO:
Programación

UNIDAD:
Aplicación de las estructuras de almacenamiento

CEAC
FP Oficial

ÍNDICE

6. APLICACIÓN DE LAS ESTRUCTURAS DE ALMACENAMIENTO

En programación es tan importante mantener una organización estricta de los datos provenientes del exterior de nuestro programa, como de los datos definidos internamente en él.

Para tal propósito, las **estructuras de almacenamiento de la información** son la mejor herramienta aliada que nos ofrecen los lenguajes de programación. En ocasiones, veremos que las variables simples capaces de almacenar un único tipo de dato no serán suficientes para conseguir que el programa cumpla sus propósitos.

Las estructuras de almacenamiento nos ofrecen una serie de objetos capaces de almacenar varios datos a la vez e incluso que puedan ser éstos de distinto tipo. Se trata de estructuras destinadas a almacenar bloques de información compleja de una manera estructurada y que permita un acceso sencillo y directo a los datos.

Cuando la información externa a nuestro programa proviene de registros o bases de datos, es una buena praxis almacenarla dentro de nuestro programa con una estructura que imita la original. Para ello, las estructuras de almacenamiento están provistas de toda una serie de características que facilitarán tanto su almacenamiento como su acceso posterior.

Por otra parte, debemos gestionar vectores de elementos (***arrays*** en Java) con un tipo de datos determinado. A veces, este tipo de datos será simple (int, char...); pero, en ocasiones, el vector puede contener elementos que son de un tipo de datos complejo, como por ejemplo objetos de una clase determinada.

También veremos un caso especial de ***arrays***, el de las cadenas de caracteres. En diversos problemas se nos plantean operaciones con cadenas de caracteres, como comparación de cadenas, búsqueda de cadenas dentro de cadenas, ordenación de caracteres, y otras operaciones usuales con las que nos podemos encontrar.

6.1 Estructuras

Una **estructura** es un grupo de elementos heterogéneos relacionados de forma conveniente con el programador y el usuario del programa. El resultado es un nuevo tipo de dato que permite una estructura organizativa más

compleja que los tipos elementales que ofrecen un lenguaje. Se trata de una combinación de varios tipos de datos, incluyendo otras estructuras que hayamos podido definir previamente.

En el ejemplo de la figura 6.1, el programa empieza definiendo una estructura con la palabra clave ***struct*** seguida de tres variables sencillas, las cuales representan sus componentes. Seguidamente, encontramos una variable "entrada" que está definida como un vector de 30 posiciones cuyo tipo de datos es esa estructura. Es decir, cada uno de los elementos del vector entrada tendrá en su contenido las tres variables de la estructura.

En el main, podemos ver cómo asignar el valor de un elemento. Para ello, situados con índice 0, se debe dar valor a todo el contenido del elemento; es decir, a todas las variables de la estructura, tal y como se ve en la figura dentro del main.

En lenguajes orientados a objetos como Java, las estructuras se solucionan con clases y objetos, pues las propiedades de los mismos cumplen perfectamente con el cometido de las estructuras.

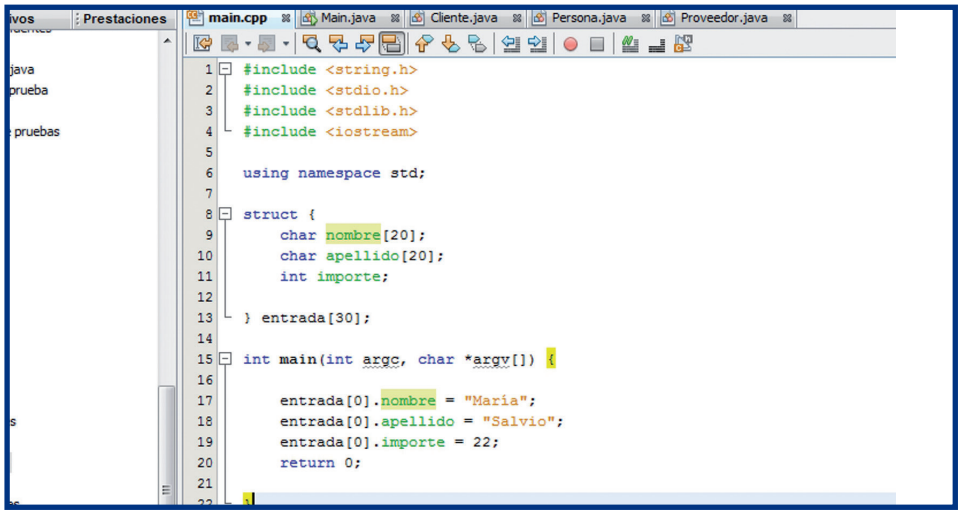


Figura 6.1
Definición de una estructura aplicada a dos variables en lenguaje C++.

6.2 Creación de *arrays*

En programación, el término inglés ***array*** (matriz o vector) define una zona de almacenamiento continuo de datos de un mismo tipo. Desde un punto de vista lógico, una matriz puede entenderse como un conjunto de elementos ordenados en serie. Las matrices son adecuadas para situaciones en las que el acceso a los datos deba realizarse de forma aleatoria e impredecible (Figura 6.2).

Todo vector o matriz se compone de un determinado número de elementos. Cada elemento está referenciado por un **índice** o posición que ocupa dentro del vector.

Figura 6.2
Un array donde el índice es la etiqueta y los valores lo que se guarda en el interior.

Existen tres formas de indexar los elementos de una matriz:

- **Indexación base-cero (0).** El primer elemento del vector recibe el índice numérico '0'. En consecuencia, el último elemento del vector recibirá un índice numérico igual al número de elementos totales menos uno. El lenguaje C es un ejemplo típico que utiliza este modo de indexación.

- **Indexación base-uno (1).** En esta forma de indexación, el primer elemento de la matriz tiene el índice '1' y el último tiene el índice igual al número de elementos totales.

- **Indexación base-n (n).** Se trata de un modo de indexación en la que el índice del primer elemento puede ser elegido libremente. En algunos lenguajes de programación se permite que los índices sean negativos e incluso también cadenas de caracteres.

Declaración de *arrays* en Java:

nombre = new <tipo>[dimensión];
Un ejemplo con un tipo simple.
int[] notas;
notas = new int[10]:

Pero también se puede aplicar a objetos, es decir, a tipos de datos complejos:

```
Persona[] personas = new Persona[10];
```

6.3 Inicialización

Como toda otra variable, cuando definimos una matriz en nuestro código, ésta se encuentra vacía. La **inicialización** de un *array* es el proceso a través del cual definimos el número de elementos que la conformarán y asignamos datos a cada uno de ellos. Existen diferentes métodos para inicializar un *array* (Figura 6.3), todos ellos el resultado de una colección de datos ordenados de manera correlativa, de la que deberemos escoger el que mejor se ajuste a nuestras necesidades:

• **Inicialización directa.** Consiste en asignar los datos para cada índice de la matriz de modo manual, esto es, asignar manualmente un valor a cada uno de los índices de la matriz.

En Java, por ejemplo:

```
<tipo> [] nombreDeTabla = { valor1, valor2, valor3... }
```

• **Inicialización dinámica.** La filosofía de la inicialización dinámica es automatizar el proceso de asignación de datos para cada uno de los índices de la matriz. Este proceso es especialmente útil cuando los datos que debemos almacenar provienen de una fuente externa al programa, como por ejemplo de un documento XML. Gracias a la ayuda de una estructura de repetición (*for, while*) seremos capaces de introducir y ordenar directamente los diferentes datos externos en la matriz.

Recuerda

La mejor manera de recorrer una tabla es la estructura de control for for(int=0;i<personas.length;i++)

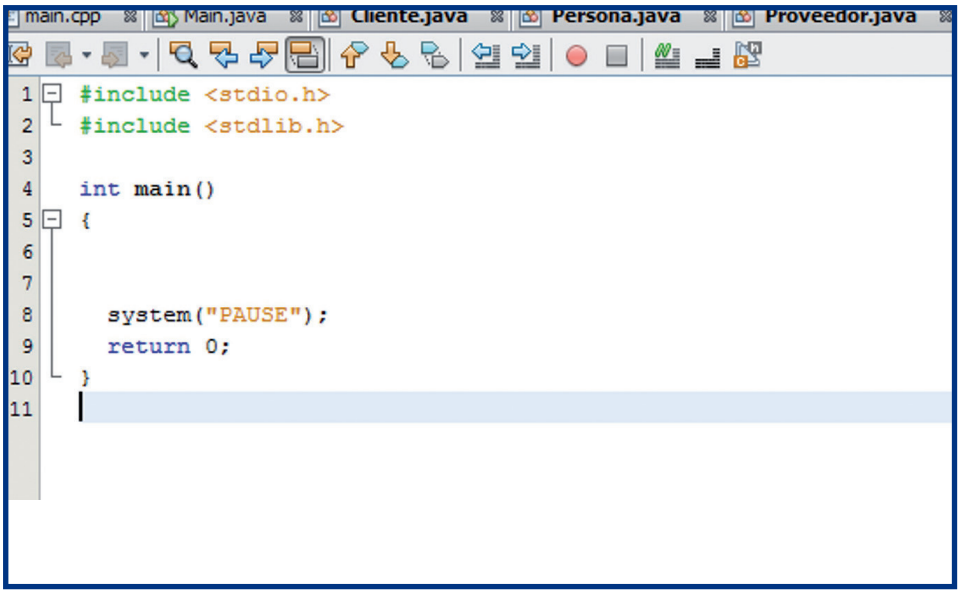


Figura 6.3
Inicialización directa de un vector de String en Java.

En Java, para referirnos a un elemento en concreto de la tabla, lo haremos como en el ejemplo siguiente que imprime el nombre del elemento 3, cuarto en realidad, de la tabla de personas.

```
System.out.println(personas[3].getNombre());
```

Se debe tener en cuenta que el elemento 3, es el de posición 4 dentro de la tabla, pues el índice empieza en 0 , por lo que, en el ejemplo anterior, saldría por consola el nombre de Sandra.

Recuerda

*La inicialización dinámica de **arrays** resulta una herramienta valiosísima en cuanto a almacenaje de datos se refiere debido al sistema de automatización que ofrece.*

6.4 Arrays multidimensionales

Los ***arrays* multidimensionales** son unas estructuras de datos capaces de almacenar valores en diferentes niveles. Los arrays que hemos visto hasta ahora almacenan valores en una única dimensión, es decir, con un índice asignado para cada una de sus posiciones y elementos. Los arrays de dos dimensiones (Figura 6.4) guardan sus valores en una estructura similar a las filas y las columnas de una tabla, por ello se requieren dos índices para acceder a cada una de las posiciones de sus datos. En un sentido amplio, podemos entender un ***array* multidimensional** como un ***array*** que contiene otro ***array*** indexado en cada una de sus posiciones (Figura 6.5).

Para referenciar un dato contenido en un *array* multidimensional deberemos indicar dos posiciones numéricas, la primera hará referencia al índice de la matriz principal y la segunda al índice de la posición del dato de la matriz contenida en él.

Figura 6.4
Un array de dos dimensiones necesita dos índices (la fila y la columna).

En Java, la declaración de un ***array*** bidimensional del tipo hoja de cálculo, es decir, con filas y columnas, se efectúa de la manera siguiente:

```
<UnTipo> tabla [][] = new <UnTipo> [unaDimensión][];
```

En este caso, sólo se sabe que los elementos serán del tipo UnTipo y que tendrá un número de filas igual a unaDimensión.

Pensemos en un hotel que tenga 4 plantas (0, 1, 2 y 3) y en cada planta tiene 5 habitaciones (de la 0 a la 4). Pequeño, pero útil para la explicación.

En cada habitación podemos tener huéspedes alojados de un tipo de datos determinado. Podríamos guardar enteros, y entonces tendríamos:

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

Y para saber el huésped de la segunda planta primera habitación podemos hacer:

```
habitacionElegida = hotel[1][0];
```

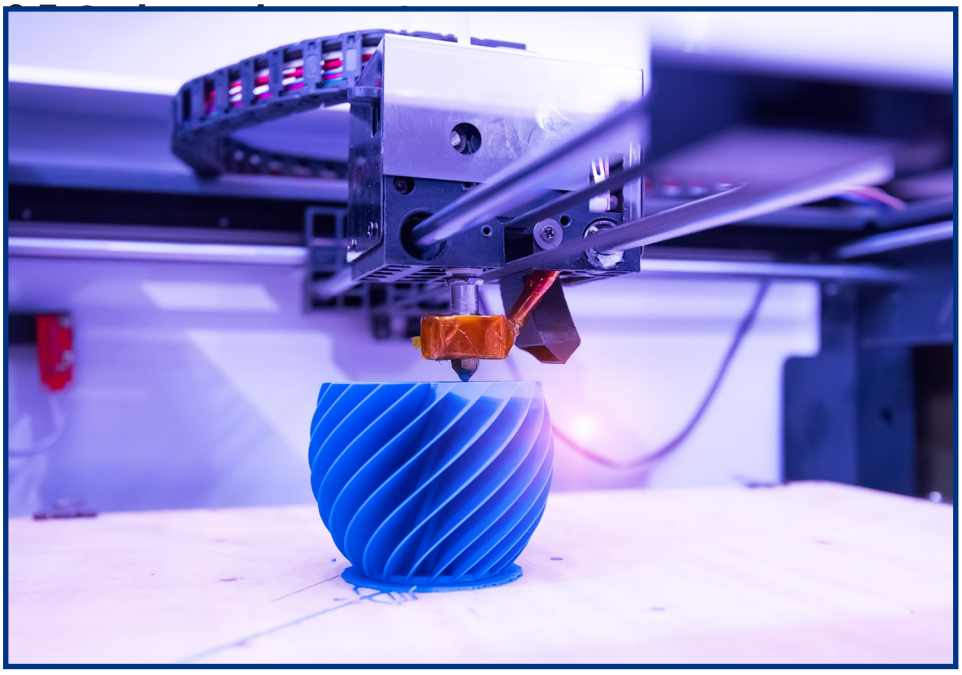


Figura 6.5
En la impresión 3D no deja de haber un array de tres dimensiones con tres índices: fila, columna y piso, donde puede haber o no material.

Las **cadenas de caracteres** (String) nos permiten almacenar conjuntos seriados de caracteres. En ese sentido, podemos entender la mayoría de datos que almacenamos en una variable de tipo String como palabras o texto en general. Sin embargo, para el compilador de un lenguaje de programación, un String es tratado como una serie de caracteres colocados en serie, a los cuales se les asigna una posición propia dentro de la variable contenedora. Así pues, las cadenas de caracteres son entendidas por el compilador del lenguaje, no como una entidad en sí, sino como un ***array*** de caracteres o letras con posiciones individuales para cada una de ellas. Del mismo modo que podemos acceder a un dato indexado dentro de una matriz, podremos también acceder a un caracter concreto dentro de una variable de tipo String indicando su posición dentro de ella.

Para saber más

*De todas las matrices multidimendionales, las de uso más común son las de dos dimensiones. Sin embargo, a medida que dotemos un **array** de mayores dimensiones, podremos conseguir efectos más complejos para nuestro programa, ya sea para almacenar datos o ya sea para representar objetos gráficos por pantalla.*

Las cadenas en Java

Declaración

En Java, una cadena de caracteres es un String. Su declaración es:

```
String str1;
```

Inicialización habitual

Caso más usual:

```
String str1 = "Hola";
```

String inicializado a partir de una cadena de caracteres

```
char t[]="ABC";
String str1 = new String(t);
```

Operaciones (métodos)

En Java, la clase String proporciona a los objetos creados con dicha clase, es decir a las variables con ese tipo, operaciones (métodos) de comparación, copia y reemplazo entre otras.

6.6 Listas

Una **lista** una estructura de almacenamiento de datos capaz de incrementar o reducir de forma dinámica el número de elementos que contiene. Al mismo tiempo, los elementos contenidos dentro de una **lista** no tienen necesidad de ser del mismo tipo (Figura 6.6). La ventaja principal de una **lista** con respecto a una matriz corriente, es que ésta puede contener tantos tipos de objetos de datos como el programador necesite. La instrucción que define una **lista** dentro del código es ***ArrayList.*** Para inicializar una **lista** podemos utilizar dos métodos, bien definiéndola como vacía y, por tanto, sin número inicial de elementos, o bien definiendo un número determinado de posiciones iniciales. Ambos sistemas son completamente válidos,

pero es recomendable que, para economizar memoria, si conocemos de antemano el número de elementos mínimos que vayamos a necesitar, lo definamos al declarar la **lista**.

En Java, la gestión de listas comprende los siguientes aspectos:

0	María
1	53
2	456,23
3	

Figura 6.6
En las listas, cada elemento puede ser de un tipo diferente, incluyendo objetos.

- **Declaración**

En Java, para poder utilizar una lista, no se puede crear un objeto de la clase List, ya que es abstracta y, por lo tanto, es necesario crear un objeto de ArrayList.

```
ArrayList losTelefonos = new ArrayList();
```

```
+ArrayList<Persona> unasPersonas = new ArraList<Persona>();
```

- **Inicialización habitual**

En cualquier caso, se emplea el método add. Por ejemplo:

```
Persona laPersona = new Persona();
unasPersonas.add(laPersona);
```

- **Operaciones (métodos)**

En Java, un objeto del tipo List, aparte de la operación añadir descrita, tiene operaciones de borrado, cuenta del número de elementos, búsqueda de un elemento, etc. Por otra parte, la lista proporciona la propiedad Iterator que se utiliza para recorrer la lista.

6.7 Colecciones

Una colección, llamada comúnmente en inglés Collection, es un sistema, implementado en Java mediante clases, que permite agrupar una serie de objetos de modo que se puede recorrer o iterar, y del cual conocemos su tamaño (Figura 6.7). Las operaciones básicas de una colección son las siguientes:

- ***add*(T).** Añade un elemento.
- ***Iterator()***. Obtiene un "iterador" que permite recorrer la colección visitando cada elemento una vez.
- ***size()*.** Obtiene la cantidad de elementos almacenados dentro de una colección.
- ***contains*(t).** Devuelve un valor booleano (cierto o falso) según si el elemento entre paréntesis (t) se encuentra dentro de la colección o no.

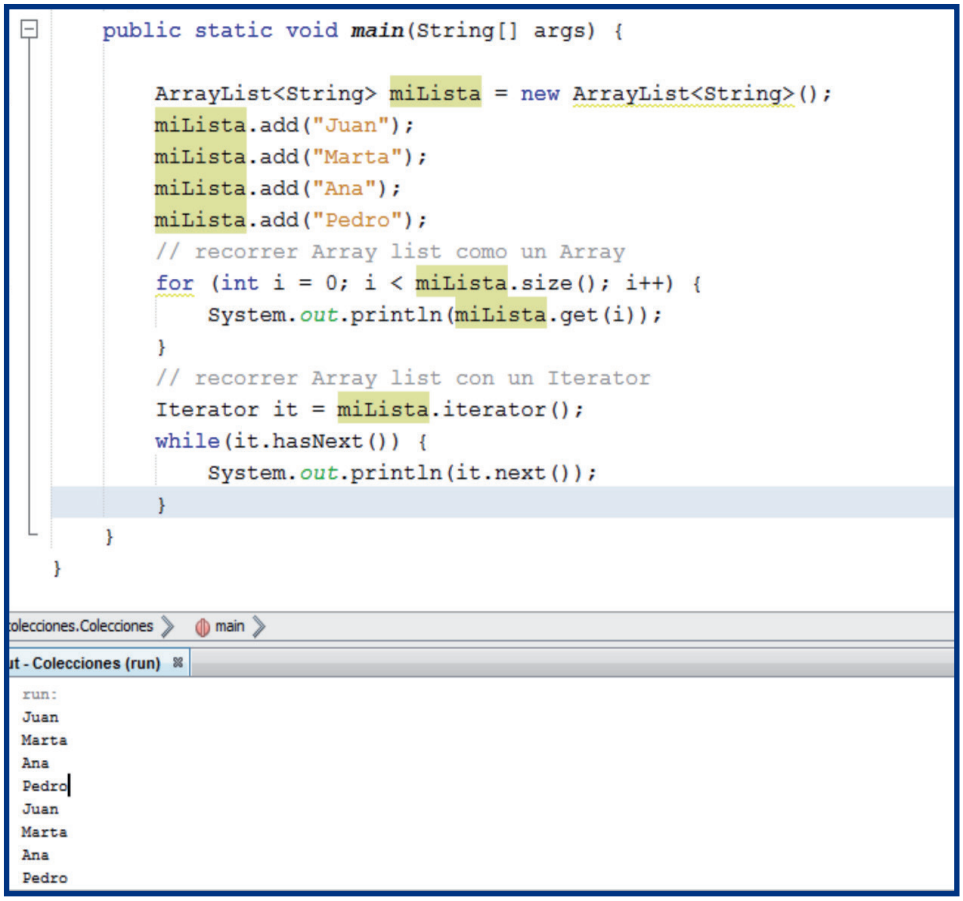


Figura 6.7
Ejemplo de uso de un ArrayList.

La figura 6.7 muestra cómo crear un ***ArrayList*** de String y cómo usar los métodos size() y get() para acceder a los elementos contenidos en la lista. También muestra cómo se puede acceder usando un iterator.

La característica principal de un objeto ***Collection*** es la de poder ser recorrido. Si bien en este punto del proceso no está definido un orden, la única manera de definirlo es usando una instrucción iteradora de repetición, mediante el método *iterator*(). Un iterador es un objeto que recorre la extensión de la colección y nos permite obtener todos los objetos contenidos en ella, invocando progresivamente un método llamado next(). Si la colección es modificable, podremos eliminar un objeto durante el recorrido mediante el método llamado remove() del iterador.

En Java la gestión de colecciones se efectúa a través de las clases heredadas de ***Collection***, como son ***ArrayList***, Vector...

Resumen

Una estructura es un grupo de elementos relacionados entre sí, que es el resultado de un nuevo tipo de dato considerablemente más complejo que los que hemos utilizado hasta ahora. Se trata de una combinación de varios tipos de datos, incluyendo otras estructuras que hayamos definido previamente. Se utilizan en lenguajes que no incorporan la POO, ya que las clases se pueden considerar una evolución de las estructuras.

Un array define una zona de almacenamiento continuo de datos de un mismo tipo. Desde un punto de vista lógico, una matriz puede entenderse como un conjunto de elementos ordenados en serie. Las matrices son adecuadas para situaciones en las que el acceso a los datos deba realizarse de forma aleatoria e impredecible.

La inicialización de un array es el proceso a través del cual definimos el número de elementos que la conformarán y asignamos datos a cada uno de ellos.

Los arrays multidimensionales son estructuras de datos capaces de almacenar valores en diferentes niveles. En ellos se almacenan los valores en una estructura similar a las filas y las columnas de una tabla.

Las cadenas de caracteres (String) nos permiten almacenar conjuntos seriados de caracteres. Para el compilador de un lenguaje de programación, un string es tratada como una serie de caracteres colocados en serie con una posición propia para cada uno de ellos. Así pues, las cadenas de caracteres son entendidas como un array de caracteres o letras.

Una lista es una estructura de almacenamiento de datos capaz de incrementar o reducir de forma dinámica el número de elementos que contiene. Las ventajas de una lista frente a un array común son la capacidad de almacenar elementos de distinto tipo en su estructura y las modificaciones en el número de elementos.

Una Collection es el concepto base del que derivan la mayoría de las estructuras de almacenamiento de información. Su característica principal es la de poder obtener datos de él y modificarlos a la vez que se recorre su extensión.

Ejercicios de autocomprobación

Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):

1. Una estructura es un grupo de elementos homogéneos relacionados de forma conveniente con el programador y el usuario del programa.

2. En lenguajes orientados a objetos, como Java, las estructuras se solucionan con clases y objetos, pues las propiedades de los mismos cumplen perfectamente con el cometido de las estructuras.

3. En programación, el término inglés **array** (matriz o vector) define una zona de almacenamiento continuo de datos de un mismo tipo.

4. Todo vector o matriz se compone de un determinado número de elementos. Cada elemento está referenciado por un índice o posición que ocupa dentro del vector.

5. Indexación base-n (n) es un modo de indexación en la que el índice del primer elemento puede ser elegido libremente. En ningún lenguaje de programación se permite que los índices sean negativos.

6. La inicialización directa consiste en asignar automáticamente un valor a cada uno de los índices de la matriz.

7. Una colección, llamada comúnmente en inglés **Collection**, es un sistema, implementado en Java mediante clases, que permite agrupar una serie de objetos de modo que se pueda recorrer o iterar, y cuyo tamaño conocemos.

Completa las siguientes afirmaciones:

8. Indexación base-cero (0). El primer elemento del vector recibe el _____ numérico '0'. En consecuencia, el último elemento del vector recibirá un índice numérico igual al número de elementos totales menos _____.

9. La filosofía de la inicialización _____ es automatizar el proceso de asignación de datos para cada uno de los índices de la _____. Este proceso es especialmente útil cuando los datos que debemos almacenar provienen de una fuente externa al programa, como por ejemplo de un documento _____.

10. Los arrays multidimensionales son unas estructuras de datos capaces de almacenar _____ en diferentes niveles. En un sentido amplio, podemos entender un array multidimensional como un array que contiene otro array _____ en cada una de sus posiciones.

Soluciones de los ejercicios de autocomprobación

Unidad 7

1. V
2. V
3. F. Una vez definida una clase como subclase, podremos utilizar y reinterpretar las propiedades heredadas, e incluso ocultarlas para que no se pueda acceder desde el exterior.
4. F. El parámetro override indica que, si existe una misma función en una subclase y su clase madre, serán las acciones definidas en la subclase las que prevalecerán.
5. V
6. F. En cuanto a las clases, la especificación abstract hace referencia a clases que no permiten su instanciación directa en el código, pero sí, en cambio, que otras clases extiendan de ellas.
7. V
8. composición, herencia, composición, herencia.
9. interfaces, clases.
10. corchetes, subclases.

ÍNDICE

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

MÓDULO:
Programación

UNIDAD:
Utilización avanzada de clases

CEAC
FP Oficial

MÓDULO:
Programación

UNIDAD:
Utilización avanzada de clases

CEAC
FP Oficial

ÍNDICE

7. UTILIZACIÓN AVANZADA DE CLASES

El estudio de las clases y su distintos usos y posibilidades dentro de la **programación orientada a objetos (POO)** es muy amplio. Ya hemos dado anteriormente una explicación global a estos conceptos.

El objetivo de esta unidad es la de ofrecer una visión más específica sobre sus características. Veremos cómo la **herencia** supone un concepto mucho más amplio que el simple hecho de compartir características entre **clases** y cómo puede ofrecernos un mayor control de nuestros programas.

Analizaremos cómo debemos entender y dotar de un sentido más específico el proceso de definición de **clases** y cómo puede ayudarnos a simplificar la estructura de nuestros códigos.

Veremos también la importancia del **anidado** y **empaquetado de clases** creando grupos organizados de objetos capaces de realizar tanto acciones compartidas como específicas, y cómo, a partir de una misma función, pueden derivarse otras funciones con unas modificaciones determinadas. Explicaremos también el uso de clases y de cómo estas pretenden representar objetos y conceptos de la vida cotidiana.

Avanzaremos en el concepto de polimorfismo y en su utilización como herramienta para desarrollar funciones que, aunque tengan el mismo identificador, su implementación es diferente según los parámetros de entrada.

La aplicación de estos conceptos, siguiendo el camino empezado en las unidades anteriores, la llevaremos a cabo a través del lenguaje de programación Java.

Las instrucciones del lenguaje, y sobre todo, las librerías de objetos que podemos utilizar, hacen del mismo una potente solución para la programación orientada a objetos, para el cometido de esta unidad: el uso avanzado de clases, la herencia y el polimorfismo.
Java no tiene herencia múltiple, sin embargo, este hecho no ocasiona ningún problema para transformar cualquier modelo de objetos en una verdadera implementación totalmente funcional, ya que podemos usar las interfaces, que obligan a un comportamiento y apoyan la transformación del modelo indicado.

7.1 Composición de clases

La **composición de clases**, también conocida como **relación asociativa**, es la relación que se establece entre dos objetos que tienen una comunicación persistente, es decir, que existe una relación de dependencia para que puedan llevar a cabo sus respectivas funciones. La composición de clases implica que uno de los dos objetos involucrados está **compuesto** por el otro. Desde el punto de vista práctico, es posible reconocer asociatividad entre dos clases A y B si la proposición **"A tiene un B"** es verdadera. Por ejemplo: "un ordenador tiene un teclado" es verdadero. Por tanto, un objeto de tipo Ordenador tiene una relación de composición con un objeto de tipo Teclado (Figura 7.1). Los objetos que componen una **clase contenedora** deben existir desde el inicio del programa. No hay posibilidad de que una clase contenedora pueda existir sin alguno de sus objetos componentes, razón por la que la existencia de estos objetos no debe ser abiertamente manipulada desde el exterior de la clase.

Siguiendo la premisa de dependencia entre dos clases, es fácil confundir el concepto de **composición** con el concepto de **herencia**, sin embargo existe una notable diferencia entre ellos. Mientras que la **composición** implica que un objeto contiene al otro, la herencia implica que un objeto deriva del otro. Es decir, según la **composición** un ordenador posee un teclado pero no es un teclado.

Figura 7.1
Un equipo informático de sobremesa representa una composición de objetos (torre, pantalla, teclado y ratón).

En la unidad anterior, concretamente en el estudio de AWT, ya hemos visto la implementación concreta de la composición en Java. Un contenerdor Frame que contiene por ejemplo una etiqueta, un cuadro de texto y un botón, que se han creado como propiedades de la ventana principal, es una implementación de composición.

Estos componentes concretos (no su clase general) no tienen razón de existir sin estar contenidos en el objeto principal. No podemos crear objetos de ellos en otras clases, son lo que son dentro de la ventana y no fuera.

```
public class AWTCounter extends java.awt.Frame implements ActionLis-
tener {

    private Label lblCount;
    private TextField tfCount;
    private Button btnCount;
...
```

7.2 Herencia

El concepto de herencia hace referencia a la relación existente entre dos objetos o clases, en la que una deriva de la otra. Mientras que la composición simplemente indica la existencia de una clase dentro de otra, y, por lo tanto, sólo se puede usar el objecto "como es (as is)", la herencia actua como cesión de propiedades entre clases y, por lo tanto, se pueden redefinir las clases heredadas a partir de las superiores. Si tomamos como ejemplo un objeto llamado Persona, en el que se definen una serie de propiedades como nombre, edad, sexo, etc., y otro llamado Empleado que deriva de él, se entiende que el objeto ***Empleado*** poseerá, además de sus propias características, todas las del objeto ***Persona***. En ese sentido, podemos pues decir que un ***Empleado*** es una ***Persona***, premisa que no se cumple en la composición entre dos clases. Toda clase que deriva de otra es llamada ***subclase***.

La herencia nos permite crear una estructura jerárquica de clases cada vez más especializada (Figura 7.2). La mayor ventaja que ofrece este proceso es que cuando se quiere especializar una clase existente no es necesario empezar desde cero, sino que basta con hacer que ésta herede las propiedades y los métodos de otra más global. Como resultado, es posible construir bibliotecas de clases que ofrecen una base que podrá ser especializada según lo creamos conveniente.

En diversos lenguajes de programación existe el concepto de herencia múltiple, que significa que una clase puede heredar de varias a la vez, adoptando de este modo propiedades y atributos pertenecientes de varías superclases.

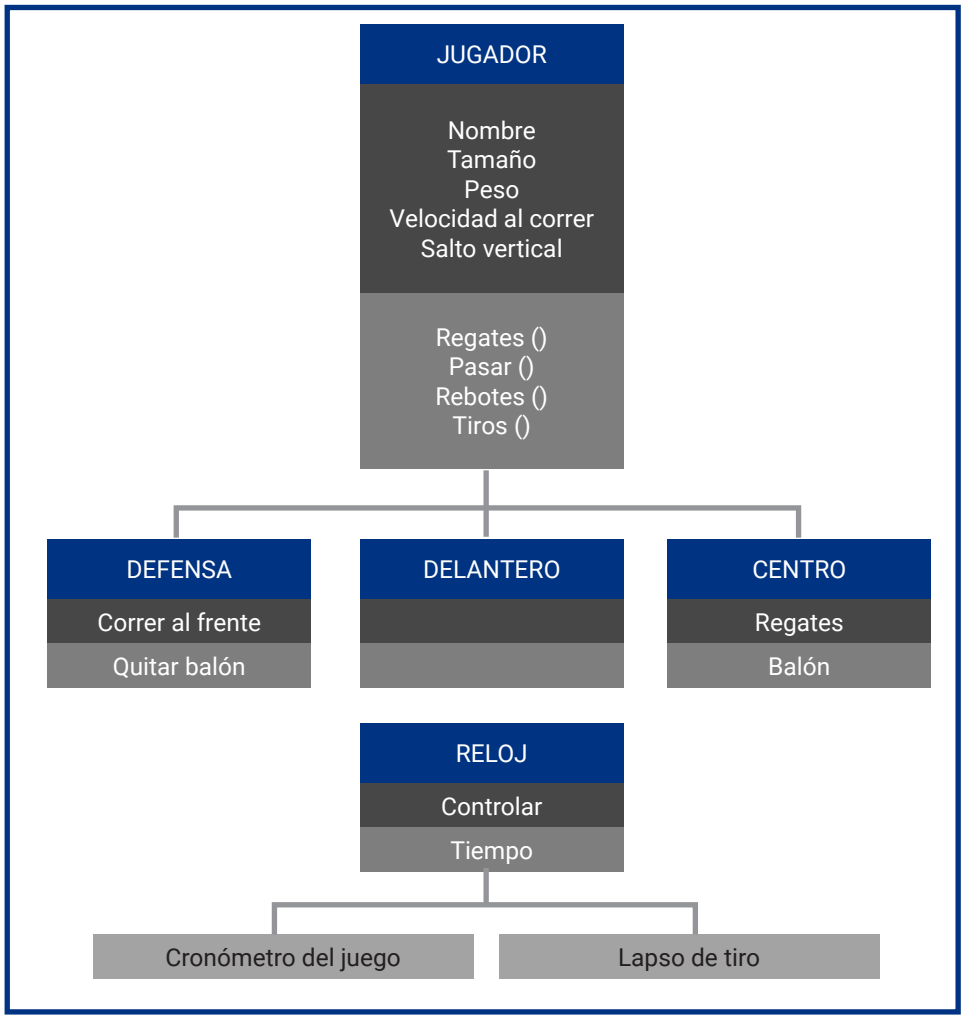


Figura 7.2
Herencia de clases de jugadores de fútbol.

En Java una clase sólo puede heredar de una a la vez, sin embargo, a su vez, puede implementar una o varias interfaces, hecho que dota al lenguaje de la posibilidad de implementar cualquier modelo de clases, por complejo que sea.

Como hemos visto en unidades anteriores, la sintaxis en Java para la herencia es:

```
class Profesor extends Persona
```

donde *Profesor* hereda las propiedades y métodos de *Persona*.

Por otra parte, podemos tener otras clases como *Alumno* y *PAS* (personal no docente) que heredan de Persona. Sin embargo, nos interesa que *Profesor* y *PAS* hereden también de la clase *Trabajador*, y en Java esto no es posible.

Recuerda

Existe una notable diferencia entre la composición y la herencia, pues mientras que la composición es un objeto que contiene al otro, la herencia es un objeto deriva del otro.

Sin embargo, podemos pensar en por qué deben heredar de Trabajador y, realizando un análisis, llegamos a la conclusión de que están obligados a fichar y a cobrar la nómina. En tal situación, se puede pensar en la clase Trabajador como una interfaz:

```
Interfaz Trabajador{
      public void fichar(Persona p){
      }
      public void cobrar(Persona p){
      }
}
```

y en tal situación, *Profesor* y *PAS* implementan esa interfaz. Es decir, están obligados a implementar sus métodos, pero cada uno fichará y cobrará según un algoritmo específico.

```
public class Profesor extends Persona implements Trabajador{
      public void fichar(Persona p){
            // Código x ....
      }
      public void cobrar(Persona p){
```

```
            // Código Y
      }
}
```

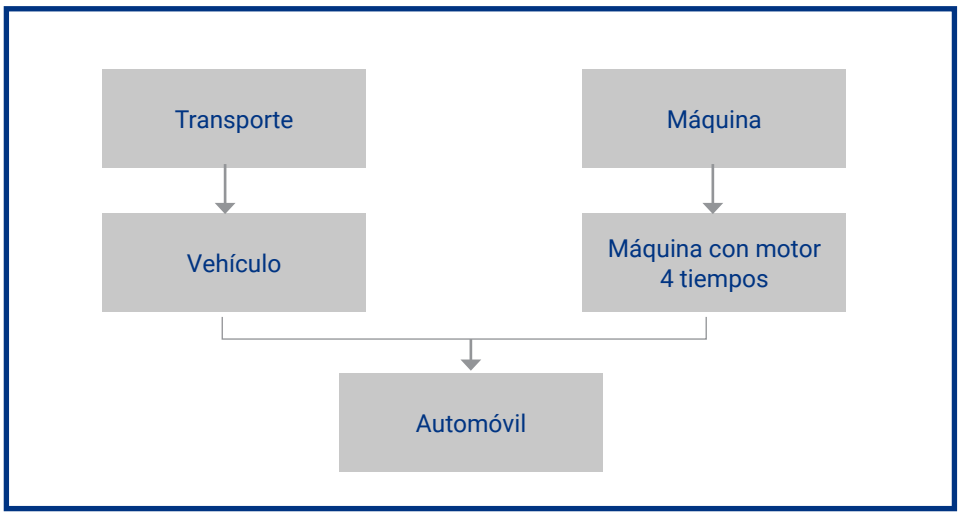


Figura 7.3
Jerarquía de clases con herencia múltiple.

7.3 Superclases y subclases

Superclase y **subclase** son los nombres que reciben las clases generales y sus clases derivadas respectivamente. Una subclase es un objeto que deriva de una clase más general y que, como tal, adquiere todas las propiedades especificadas en ella. Siguiendo el ejemplo anterior en el que proponíamos dos clases llamadas *Persona* y *Empleado*, podemos definir que Persona se refiere a la clase madre y *Empleado* a una clase derivada; es decir, *Persona* es la superclase de *Empleado* y *Empleado* es una subclase de *Persona*.

La creación de **subclases** nos permite crear nuevas clases personalizadas y más específicas tomando los elementos necesarios de una clase ya existente. De este modo, nos ahorramos tener que volver a escribir todas las propiedades, métodos y eventos que ya posee la clase madre. Una vez definida una clase como subclase, podremos utilizar y reinterpretar las propiedades heredadas, e incluso ocultarlas para que no se pueda acceder desde el exterior. En programación, cuando una clase deriva de otra, se dice que la extiende, y la instrucción utilizada para indicarlo es ***extends*** seguido del nombre de la clase madre (Figura 7.4).

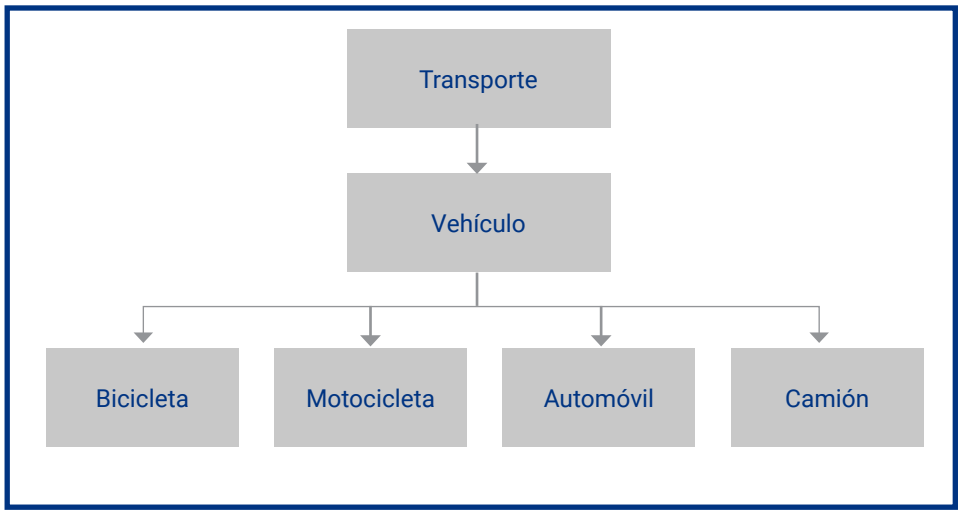


Figura 7.4
Vehículo es superclase de motocicleta y bicicleta y, a su vez, subclase de transporte.

7.4 Clases y métodos abstractos y finales

Abstracto (en inglés *abstract*) y **final** (en inglés *final*) son dos tipos específicos de declaración de clases y métodos de clase. En cuanto a las clases, la especificación abstract hace referencia a clases que no permiten su instanciación directa en el código, pero sí en cambio que otras clases extiendan de ellas. Los métodos declarados como abstract son funciones definidas destinadas a realizar acciones directas. La diferencia básica entre un método o función abstracta y otro común es su estructura o modo de definición. Mientras que en una función común se especifican las acciones a realizar con un bloque de código delimitado entre corchetes, en una función abstracta sólo se deja preparado el método, pero sin código, para que sea

Para saber más

Una clase abstracta puede tener definidos datos y métodos estáticos. Para utilizar una de estas propiedades, sólo hay que referenciar la clase a la que pertenecen. Por ejemplo, ClaseAbstracta.metodoEstatico(), como se haría con cualquier otra clase.

implementada en las subclases. Una clase debe declararse abstracta sólo cuando contenga métodos abstractos definidos en su estructura (Figura 7.5). Este mecanismo es similar a las funciones virtuales puras del C++ que hacen que la clase sea abstracta.

La especificación ***final*** se usa para definir una clase de la que no podrá derivar ninguna otra. Se trata de un parámetro destinado a indicar que una clase representa el final de la jerarquía parcial o total de una biblioteca de clases.

```
1   abstract class FiguraGeometrica {
2       ...
3       abstract void dibujar();
4       ...
5   }
6
7   class Circulo extends FiguraGeometrica {
8       ...
9       void dibujar(){
10          //código para dibujar círculo
11      }
12  }
13
```

Figura 7.5
Ejemplo de clase abstracta con método abstracto.

7.5 Sobreescritura de métodos

Cuando definimos una subclase, ésta hereda todas las propiedades y métodos de su clase madre de manera automática. Sin embargo, en ocasiones, puede ocurrir que una subclase esté destinada a realizar acciones definidas en su clase madre pero con ciertas variaciones. Dicha situación resulta de una interferencia entre ambas funciones, y para evitarlo debe utilizarse el parámetro ***override*** (sobreescribir) para aquella función definida en la subclase (Figura 7.6). El parámetro override indica que, si existe una misma función en una subclase y su clase madre, serán las acciones definidas en la subclase las que prevalecerán.

Pongamos como ejemplo una clase principal llamada Animal, en la cual se definen las acciones básicas que realizan todos los animales sea cual sea su especie (comer, beber, correr, etc.). Los métodos definidos en la superclase son heredados por todas las subclases que de ella se deriven, pero sólo definen acciones genéricas, es decir, se sobreentiende que todos los animales beben agua, aunque no todos lo hacen de la misma manera. Un perro o un gato utilizan un mismo método, pero para el caso de un elefante, por ejemplo, es distinto. Así pues, en la definición del método *beber* de la clase *Elefante* utilizaríamos el parámetro override para definir las modificaciones necesarias.

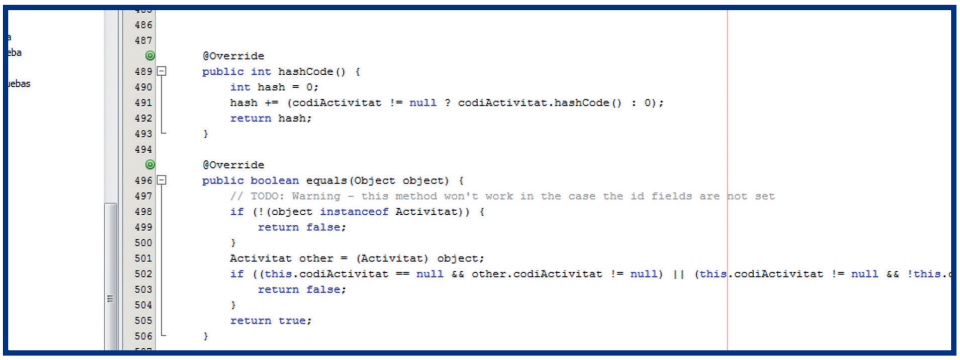


Figura 7.6
Ejemplo de subclase donde los métodos que se reescriben se anotan con @ override.

En Java, como se ha explicado, definir que una clase hereda de otra se realiza en la declaración con la palabra reservada extends:

```
classe Profesor extends Persona{
}
```

Una clase declarada así hace que, aunque no estén en el código, todas las propiedades y métodos de Persona estén incluidas en Profesor. Imaginemos un método de Persona que muestra el nombre de la persona a partir de su NIF (por ejemplo, para mostrarlo en una ventana de una aplicación). La clase *Profesor* puede tener una propiedad añadida con el departamento y nos interesa que en dicha aplicación también se muestre. En tal situación, el método de Persona sería simplemente mostrar el nombre y el método de Profesor reescribiría el de Persona, llamando a éste mediante la sentencia super.metodo (parámetros) y, además, mostrando el departamento.

7.6 Constructores y herencia

Como hemos definido anteriormente, los **constructores** tiene el mismo nombre que las clases. El **constructor** funciona al mismo tiempo como función principal e identificador para una clase. Recordemos que, para instanciar una clase en nuestro código, sólo necesitamos definir un dato para almacenarla y llamar a su **método constructor** para crearla. De ese modo, podemos importar en nuestro código cualquier tipo de clase, sin embargo este proceso no siempre tiene sentido. Siguiendo el ejemplo de una clase madre llamada *Animal*, nos damos cuenta de que, al instanciarla en el código, poco podemos hacer con ella, ya que, por norma general, los datos y funciones que contiene son genéricos, es decir, poco específicos. Es aquí donde toma importancia la herencia. Ésta sirve no sólo para transmitir propiedades entre clases, sino también para dotar de sentido a las acciones definidas en ellas. Que un animal puede beber y comer es evidente, pero no es representable si no asociamos dichas acciones a algún animal en concreto. Así pues, gracias a la herencia, al crear una clase llamada *Perro*, que deriva de otra llamada Animal, lo que en realidad estamos haciendo es

Recuerda

*Para evitar la interferencia entre las funciones definidas en las subclases y las superclases, deberemos utilizar el término **override** para indicar las modificaciones que incluirá la función que imperará de las dos.*

dotar de sentido a las acciones definidas en la clase Animal. La estructura de una subclase es idéntica a la de cualquier clase, el único factor que varía entre los distintos lenguajes de programación es el modo de definir de qué clase hereda las propiedades.

En Java, una clase hereda de la superclase todas las propiedades y todos los métodos excepto los constructores.

Como hemos aprendido, en Java podemos omitir un constructor. Sin embargo, esto no quiere decir que no exista, sino que Java lo creará en tiempo de ejecución, aunque no haga nada explícito.

Si en la superclase se añade un constructor en el código, en su diseño, la subclase no heredará el mismo, pero sí se puede codificar uno que llame al de la superclase con la palabra reservada super.

7.7 Acceso a métodos de la superclase

Tal y como estipulan las directrices de la herencia entre clases, los métodos definidos en una clase madre o superclase, pueden ser utilizados por todas las clases que de ella se deriven. Para ello, el requisito principal es que toda la clase que queramos definir como global para todas las subclases sea de tipo público (***public***). Con la especificación ***public*** podremos acceder a una función desde puntos externos de la clase en la que ha sido definida. Para el caso de las clases, la filosofía es la misma, sólo que conlleva la característica de que la función no sólo puede ser accedida desde una subclase si no que, además, dicha función pasa a pertenecerle. Al definir una clase llana, si introducimos una función pública en ella, ésta podrá ser invocada sólo al instanciar la clase en el código. Sin embargo, tratándose de una subclase que hereda de una clase más general, la instanciación ya no será necesaria, ya que la función pasará a formar parte activa de ella. Dicho proceso puede repetirse con todas las subsiguientes clases que definamos a partir de ella, proporcionándonos una estructura de organización sólida que favorecerá el control de nuestros códigos. Así pues, tomando los ejemplos propuestos anteriormente, acciones como *correr*, *beber*, *comer* o *mover*, definidos en una clase llamada *Animal*, deberán ser públicos para poder acceder desde todas sus subclases.

En Java, tal y como se ha mencionado con los constructores, se puede reescribir un método en la subclase. Dicho método puede reescribirse totalmente, o bien, aprovecharse de lo ya implementado en el método de la clase de la cual hereda llamándolo. Esta llamada se efectuará con la palabra reservada super (super.metodoY(xxx), donde metodoY es el método de la superclase a la que necesitamos llamar) (Figura 7.7).

```java
public class Proveedor extends Persona {
    public Double importeCompras = 0.;
    public static Double importeTotal = 0.;
    public int a;

    public Proveedor(String nombre, String telefono) {
        super(nombre, telefono);
    }

    public Proveedor(String nif){
        super(nif);
    }
}
```

Figura 7.7
Ejemplo de constructor con llamada al de la superclase con la palabra reservada "super".

7.8 Polimorfismo

En programación orientada a objetos, en ortodoxia, el **polimorfismo** se refiere a la capacidad que poseen las clases derivadas de una antecesora (subclases) para utilizar una misma acción de forma distinta. Tomemos como ejemplo dos clases llamadas *Pez* y *Ave* derivadas de la superclase *Animal*. La clase *Animal* tiene definido el método abstracto mover, el cual se implementará de forma distinta en cada una de las subclases, ya que peces y aves se mueven de forma diferente (Figura 7.8). El concepto de **polimorfismo** puede aplicarse tanto a funciones como a tipos de datos.

```
using namespace std;
class figura{
   private:
        float base;
        float altura;
   public:
        void captura();
        virtual unsigned float perimetro()=0;
        virtual unsigned float area()=0;
};
class rectangulo: public figura{
   public:
        void imprime();
        unsigned float perimetro(){return 2*(base+altura);}
        unsigned float area(){return base*altura;}
};
class triangulo: public figura{
   public:
        void muestra();
        unsigned float perimetro(){return (sqrt(base^2+altura^2))+altura+base;}
        unsigned float area(){return (base*altura)/2;}
};

void figura::captura()){
   cout<<"CÁLCULO DEL ÁREA Y PERÍMETRO DE UN RECTÁNGULO Y UN TRIÁNGULO RECTÁNGULO" <<endl;
}
```

Figura 7.8
Ejemplo de cómo diferentes clases definidas en lenguaje C++ usan de manera distinta los atributos (datos) y métodos (funciones) definidos en su clase madre llamada figura.

Existen dos tipos de **polimorfismo**:

- **Polimorfismo estático.** Cuando disponemos de un mismo método de varias formas, es en el tiempo de compilación cuando se decide cuál de ellos se usará. En este caso será el número y los tipos de parámetros los factores que permitirán al compilador seleccionar el método.

- **Polimorfismo dinámico.** En una jerarquía de clases, una referencia a un objeto de una clase puede serlo también a otras subclases, por lo que en el tiempo de ejecución se ejecutará un método u otro según a qué tipo de objeto este referenciando.

En la teoría ortodoxa de la programación orientada a objetos, el **polimorfismo** es la capacidad de que varios métodos tengan el mismo identificador pero implementen funciones diferentes.

En la práctica, en los lenguajes de programación, esto se consigue haciendo que los parámetros de entrada sean diferentes. El valor devuelto por la función puede ser igual o diferente, pero esto no hará que sea polimórfico.

Java soporta esta definición y esta característica es usada habitualmente. Por ejemplo, si pensamos en la función System.out.println(zzz), es posible que no hayamos caído en la cuenta de que es polimórfica, y lo podemos comprobar yendo a su definición: veremos que se trata de varias funciones. Al usarla, le hemos pasado parámetros diferentes sin darnos cuenta, un string normalmente, pero también un Integer, e incluso le podríamos pasar cualquier tipo de objeto (no se puede asegurar que imprimiría). Al pasarle diferentes tipos de datos, realmente se han ejecutado funciones diferentes, que se llaman igual, que tienen el mismo identificador, pero que están implementadas de forma diferente, aunque al final impriman lo que le hemos pasado, necesariamente son diferentes.

Muchas veces, se habla también de polimorfismo referido a la sobrecarga de métodos, es decir, a cuando reescribimos un método de una superclase en la clase que hereda. Esto también está aceptado, aunque no se desarrollará en este apartado porque ya está descrito en apartados anteriores.

Resumen

La composición de clases, también conocida como relación asociativa, es la relación que se establece entre dos objetos que tienen una comunicación persistente, es decir, que existe una relación de dependencia para que puedan llevar a cabo sus respectivas funciones.

La composición de clases o relación asociativa entre clases es una relación de pertenencia de un objeto a otro. Aunque el objeto que pertenece al otro puede crearse sin él, no tiene sentido por sí solo en el conjunto. Por ejemplo, un teclado y el ordenador.

En cambio, la herencia es una relación donde una subclase deriva de una superclase, obteniendo las propiedades y los métodos de ella. Por ejemplo, profesor puede heredar de la superclase persona.

Una superclase puede ser abstracta, es decir, no podrá instanciarse y sólo podrá heredarse. En este sentido, sus métodos pueden ser abstractos, con lo que sólo estarán declarados en la superclase y su implementación debe codificarse obligatoriamente en cada subclase. La clase de la cual no se puede heredar debe declararse como final.

Las subclases heredan las propiedades y métodos de la superclase, pudiendo sobrescribir los métodos de la superclase, a los cuales también se puede acceder mediante el uso de la palabra reservada super. Los métodos constructores no se heredan.

El concepto de polimorfismo es un término general de la programación orientada a objetos que corresponde a la posibilidad de que diversos métodos se denominen igual, pero para ello deben cambiar algo en los parámetros de llamada. Este concepto se extiende cuando se habla de herencia porque un método heredado se puede reescribir, aún sin cambiar los parámetros.

Ejercicios de autocomprobación

Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):

1. La composición de clases, también conocida como relación asociativa, es la relación que se establece entre dos objetos que tienen una comunicación persistente, es decir, que existe una relación de dependencia para que puedan llevar a cabo sus respectivas funciones.

2. En diversos lenguajes de programación existe el concepto de herencia múltiple, que significa que una clase puede heredar de varias a la vez, adoptando, de este modo, propiedades y atributos pertenecientes de varias superclases.

3. Una vez definida una clase como subclase, podremos utilizar y reinterpretar las propiedades heredadas, pero no podremos ocultarlas.

4. El parámetro override indica que, si existe una misma función en una subclase y su clase madre, serán las acciones definidas en la clase madre las que prevalecerán.

5. La composición implica que un objeto contiene al otro, y la herencia implica que un objeto deriva del otro.

6. En cuanto a las clases, la especificación abstract hace referencia a clases que permiten su instanciación directa en el código, pero no que otras clases extiendan de ellas.

7. En programación orientada a objetos, en ortodoxia, el polimorfismo se refiere a la capacidad que poseen las clases derivadas de una antecesora (subclases) para utilizar una misma acción de forma distinta.

Completa las siguientes afirmaciones:

8. Siguiendo la premisa de dependencia entre dos clases, es fácil confundir el concepto de _____ con el concepto de _____, sin embargo existe una notable diferencia entre ellos. Mientras que la _____ implica que un objeto contiene al otro, la _____ implica que un objeto deriva del otro.

9. En Java una clase solo puede heredar de una a la vez; sin embargo, a su vez, puede implementar una o varias _____, hecho que dota al lenguaje de la posibilidad de implementar cualquier modelo de _____ , por complejo que sea.

10. Mientras que en una función común se especifican las acciones a realizar con un bloque de código delimitado entre _____, en una función abstracta solo se deja preparado el método, pero sin código, para que sea implementada en las _____.

Soluciones de los ejercicios de autocomprobación

Unidad 7

1. V
2. V
3. F. Una vez definida una clase como subclase, podremos utilizar y reinterpretar las propiedades heredadas, e incluso ocultarlas para que no se pueda acceder desde el exterior.
4. F. El parámetro override indica que, si existe una misma función en una subclase y su clase madre, serán las acciones definidas en la subclase las que prevalecerán.
5. V
6. F. En cuanto a las clases, la especificación abstract hace referencia a clases que no permiten su instanciación directa en el código, pero sí, en cambio, que otras clases extiendan de ellas.
7. V
8. composición, herencia, composición, herencia.
9. interfaces, clases.
10. corchetes, subclases.

ÍNDICE