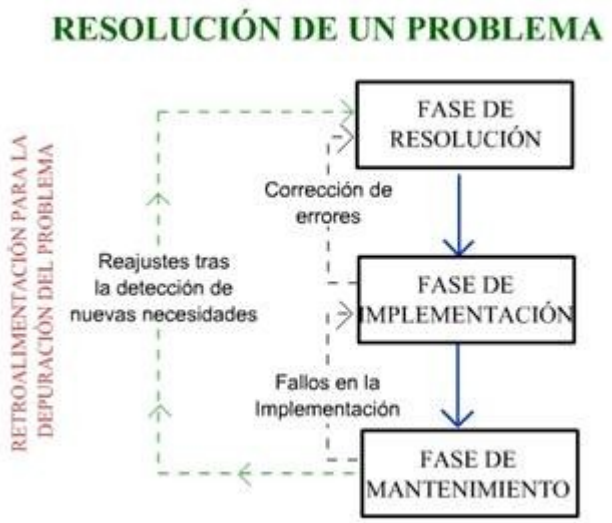
Programación en Lenguajes Estructurados (Grupo B)

Indice

UNIDAD 1: Pasos para la resolución de un problema

Pasos para la resolución de un problema.

Intenta dividir el trabajo para que pueda ser abordado por diferentes personas o grupos de trabajo y establecer una forma sistemática de establecer el trabajo para garantizar que se hace correctamente.



- **Fase de resolución:** "traducir" esa solución para que sea comprensible y ejecutable por un ordenador, corregir los errores que contenga y probar que su funcionamiento es el correcto.
- **Fase de implementación:** codificar el algoritmo, compilarlo y probarlo
- **Fase de mantenimiento:** manterner actualizado el programa, hacer ajustes, etc...

Resolución del problema

Tratamos de acercarnos al problema, definirlo correctamente, e idear una forma de resolverlo, sin conecer el lenguaje concreto en el que se va a desarrolar.

- **Análisis del problema:** Consiste en **estudiar el problema**, asegurándonos de que lo entendemos bien, de que hemos tenido en cuenta todos los casos y situaciones posibles. Es necesario dar una definición lo más exacta posible del mismo, identificando qué tipo de información se debe producir y qué datos o elementos dados en el problema pueden usarse para obtener la solución.
- **Diseño del algoritmo:** La idea es describir la solución al problema (algoritmo) con la claridad suficiente como para que trasladar la solución a cualquier lenguaje de programación concreto sea inmediata. Es frecuente usar la téncica del “divide y vencerás” o “diseño descente” que consiste en dividir el problema en subproblemas.
- **Verificación manual del algoritmo:** consiste en probrarlo con un conjunto de datos que incluyan todos los casos posibles, includos los menos frecuentes. Si nos damos cuenta de algún error, nos puede llevar a volver a cualquiera de estas fases anteriores, bien añadiendo o modificando especificaciones, o bien proponiendo cambios en la manera de resolver el problema.

Fase de Implementación

Consiste en traducir del lenguaje algorítmico a un lenguaje de programación concreto.

- **Codificación:** consiste en ir siguiendo el algoritmo, y escribiendo con la sintaxis de ese lenguaje las sentencias o instrucciones que hacen lo que se indica en el algoritmo.
- **Traducción a código máquina:** los lenguajes que normalmente se utilizan (Java, C, etc..) son llamados lenguajes de alto nivel, estos tienen que ser modficados y pasado a un lenguaje que entienda el ordenador (código binario). Estos programas se llaman traductores y hay dos tipos: los compiladores y los intérpretes.
- **Compiladores:** un compilador es un traductor que pasa un fichero de texto (en un

lenguaje concreto) llamado **código fuente** a **código máquina**. El compilador analiza el texto y devuelve una serie de errores que deben ser corregidos, a esto se le llama **depuración de errores**. Una vez que no existan errores crea el programa en código máquina que se llama **código objeto**.

- **Interpretes:** Un intérprete ejecuta el código fuente instrucción tras instruccion. Así en cada la traduce a código máquina en memoria y la ejecuta, sin guardar la traducción.
- **Depuración:** consiste en resolver los posibles errores que existan en nuestro programa.
- **Pruebas:** Consiste en probar el funcionamiento del programa con un conjunto de datos o suficientemente significativos, incluyendo valores extremos o raros. Cualquier fallo detectado nos puede llevar a replantear cualquiera de las etapas anteriores. Si es posible las pruebas las deben hacer personas ajenas al desarrollo.

Fase de explotación y mantenimiento

Se realizan tareas de mantenimiento de la aplicación, para corregir errores, introducir mejoras, adaptaciones, etc.

- **Ajustes:** Incluso después de haber superado con éxito todas las pruebas realizadas pueden aparecer pequeños fallos. Esto nos llevará a replantear la aplicación desde la fase que sea necesaria.
- **Mejoras:** de rapidez, eficiencia, manejo, etc... Esto nos llevará a replantear la aplicación desde la fase que sea necesaria**.**
- **Adaptaciones:** son situaciones nuevas, no previstas inicialmente, que requieran incluir funcionalidad nueva en nuestra aplicación.

UNIDAD 2: Datos: Tipos y características

Concepto de dato

Podemos definir **dato** como un conjunto de símbolos que representan valores, hechos, objetos o ideas de forma adecuada para ser tratados. Un dato puede ser un carácter leído desde teclado, un número, la información almacenada en un CD... La información tiene que organizarse y estructurarse de forma adecuada para que pueda almacenarse, procesarse y recuperarse de la forma más eficiente posible.

Tipos de dato

Dos tipos:

- **Constante**: se escribe su valor una vez y no se puede escribir en la posición de memoria que ocupan
- **Variable**: se puede cambiar tantas veces como se quiera.

Al nombre de la variable lo llamamos **identificador**. Cada variable o constante lleva asociado un **tipo de dato**. Un tipo de dato no es más que una especificación de los valores que son válidos para esa variable y de las operaciones que se pueden realizar con ellos.

Tipos simples de datos

Los tipos simples de datos son los datos que suele proporcionar el lenguaje de programación, y que por eso de ellos conocemos:

- Cómo se representan internamente (en memoria).
- El tamaño exacto que ocupan.
- Los operadores que define el lenguaje para ellos.
- Que están disponibles sin necesidad de definir nada por parte del usuario.

Son un conjunto mínimo de tipos de datos, de forma que a partir de ellos se construirán los demás tipos de datos, que por ello recibirán el nombre de datos estructurados.

Los datos primitivos en Java son:

- **boolean**: Permite representar valores lógicos; Verdadero (V) o Falso (F).
- **char**: Permite representar un símbolo o carácter UNICODE de 16 bits.
- **byte**: Entero de 8 bits con signo (representado en complemento a dos). Su rango de valores va desde -128 (-2^7) a +127 ($+2^7-1$).
- **short**: Entero de 16 bits con signo (complemento a dos). Rango de valores entre -32.768 (-2^{15}) y +32.767 ($+2^{15}-1$).
- **int**: Entero de 32 bits con signo (complemento a dos). Rango de valores entre -2.147.483.648 (-2^{31}) y +2.147.483.647 ($+2^{31}-1$).

- **long**: Entero de 64 bits con signo (complemento a dos). Rango de valores entre -2^{63}y $+2^{63}-1$.
- **float**: Número real (en coma flotante) de 32 bits, utilizando la representación IEEE 745-1985.
- **double**: Número real (en coma flotante) de 64 bits, utilizando la representación IEEE 745-1985.

Tipo Entero

Los tipos **numéricos** son tipos básicos que nos permiten representar valores de tipo numérico. Los datos de **tipo entero** permiten representar números enteros (sin decimales) tanto positivos como negativos, los enteros en java son: byte, short, int y long.

No obstante, actualemente la cantidad de memoria despreciada entre tipos int y byte es mínima, por lo que no va acusar el rendimiento del ordenador. Utilizaremos normalmente **int.** Sin embargo definiremos el tipo **long** sólo si es necesario.

Tenemos tener en cuenta que todos los enteros:

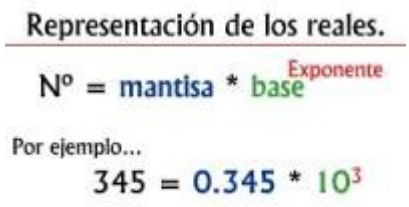
- Los **operadores** son: suma, resta, división entera (sin sacar decimales), multiplicación y operación resto-módulo (el resto de la división entera).
- Sea cual sea el tipo elegido, siempre habrá números enteros que no se podrán representar, ya que los tipos son finitos.
- Tienen una **"aritmética circular"**, de forma que si al número más grande que se puede representar para un tipo entero le sumo 1, no se produce ningún error, sino que se obtiene el número más pequeño (más negativo) que se puede representar para ese tipo. Al restar igual.

Tipo Real

Al conjunto de todos los números con decimales, ya sea un número finito o infinito de ellos, se le llamaba **reales.** Mientras más bits usemos, podremos representar números:

- Más grandes en valor absoluto (tanto positivos como negativos)
- Con mayor precisión (con más decimales exactos)

Los números reales se representan en coma flotante. Un número se expresa como: Nº = mantisa * $Base^{exponente}$. En concreto, sólo se almacena la mantisa, que son las cifras decimales significativos, y el exponente al que va elevada la base.



Existen problemas al representar los números reales al despreciar valores por que el ordenador almacena datos finito (redondeando valores).

- No siempre se cumple la propiedad asociativa ni la distributiva
- Tambien ocurren algunos problemas de incoherencia en expresiones. (X+Z)-X = no es Z
- Cuando obtenemos un resultado mayor o menor a un número representable se produce un error de **overflow** (desbordamiento por arriba o por grande) que hace que falle el programa.
- Si al operar obtenemos un resultado más próximo a cero que el número más cercano a cero que se puede representar (tanto en positivo como en negativo), se produce un error de **underflow** (desbordamiento por abajo o por pequeño) que también hace que falle el programa.

La mayoría de los lenguajes definen también como tipos básicos (o simples o primitivos), más de un tipo de números reales, que se diferencian en el número de bits usados para la representación, y por tanto en la precisión de los números representados. En java hay dos:

- **float**: 32 = 4 bytes
- **double**: 64 = 8 bytes

Los **operadores** disponibles para los tipos reales son las operaciones matemáticas usuales: suma, resta, multiplicación, división real (con decimales)

Tipo Boolean

Todos los lenguajes incluyen un **tipo de datos lógicos o booleanos**.Son datos que sólo pueden tomar dos valores distintos, (verdadero o falso, si o no, 0 ó 1)

Los operadores habituales para este tipo de datos son la conjunción lógica o Y-lógico (AND), la disyunción lógica u O-lógico (OR) y la negación lógica o NO-lógico (NOT).

Tipo Caracter

Los datos de tipo carácter representan elementos individuales del conjunto de caracteres usado como código de entrada-salida. El código más usual es el **ASCII**, que usa 8 bits (1 byte) para representar cada carácter, hasta 256 distintos. Java ha incorporado como alfabeto el código **Unicode**, más reciente que el código ASCII, y que usa 16 bits para representar cada carácter, por lo que puede representar hasta 65.536 caracteres distintos. En Java, el tipo carácter se llama ***char***, y el alfabeto o código usado es Unicode

En Java no existe el tipo fecha como un tipo básico, aunque existen clases que definen toda una amplia gama de datos estructurados para la representación de fechas y operaciones posibles sobre fechas.

Datos Estructurados

Además de los tipos simples, o básicos o primitivos, la mayoría de los lenguajes permiten usar una serie de estructuras de datos algo más complejas, que se construyen a partir de otros tipos, que pueden ser los propios tipos básicos o primitivos o ser a su vez tipos estructurados, o tipos definidos por el usuario.

Tipo Cadena de caracteres

Una cadena de caracteres es una sucesión de caracteres (letras, números y demás caracteres del alfabeto del lenguaje). Es muy habitual que se delimiten mediante comillas ("). Para poder mostrar, por ejemplo, una comilla (") dentro de la cadena y no tener problemas con las comillas que la delimitan, se usan **secuencias de escape**.

Algunas operaciones comunes:

- **Concatenar dos cadenas de caracteres para formar una nueva**. En java se usa el operador +
- **Obtener una subcadena a partir de otra**. Ej: "El empleado encargado".substring(4,12)
- **Obtener el carácter de una posición de la cadena**. Ej: "Empleado".charAt(2)
- **Buscar una subcadena o un carácter dentro de la cadena, y saber en qué posición se encuentra.**
- **Calcular la longitud de una cadena** Ej: "Empleado".legth()
- **Comprobar si empieza o termina con una secuencia de caracteres.** Ej: "Empleado".startsWith("Em") y "Empleado".endsWith("a")
- **Reemplazar parte de una cadena por otra.** Ej: "El empleado encargado".replaceAll("e","X")

Tipo Array

Un ***array*** es un conjunto de variables o registros del mismo tipo que puede estar almacenado en memoria principal o en memoria auxiliar en posiciones consecutivas. Cada posición tiene un número asociado, **índice**, que indica la posición del elemento en el array

- Los arrays de 1 dimensión se denominan ***vectores***,
- los de 2 o más dimensiones se denominan ***matrices***.

En Java, para **definir un array**, se pone el tipo de los elementos individuales, seguido de un corchete por cada dimensión que queramos darle, y del nombre del array. Posteriormente habrá que dimensionarlo, indicando cuantos elementos va a tener, y posteriormente se usará para almacenar valores con algún propósito.

```
...
int [ ][ ] arrayBidimensionalDeEnteros;
arrayBidimensionalDeEnteros = new int[3][4];
arrayBidimensionalDeEnteros[0][0]=23;
...
arrayBidimensionalDeEnteros[2][3]=376;
```

Tipo Registro

Un **registro** es una estructura de datos formada por yuxtaposición de elementos de distinto tipo que contiene información relativa a un mismo ente u objeto. A cada uno de los elementos que componen el registro se les llama ***campos***. Los campos aparecen en un orden determinado y se identifican por un nombre. Para definir el registro, hay que darle un nombre y un tipo a cada campo. El tipo de cada campo puede ser una estructura de datos a su vez.

Tipo Fichero o Archivo

Un **archivo** es un conjunto de información sobre un mismo tema, tratada como unidad de almacenamiento y organizada de forma estructurada para la búsqueda y recuperación de un dato individual. Es por tanto la estructura que necesitamos usar para poder guardar los datos e informaciones en soportes de almacenamiento masivo o permanente. Habitualmente los **ficheros o archivos** están compuestos por una colección de registros homogéneos. Las operaciones habituales son:

- Creación del fichero.
- Inserción de un registro
- Eliminación o borrado de un registro
- Consulta de uno, varios o todos los registros.
- Modificación o actualización de un registro.
- Borrado del fichero.
- Búsqueda y localización de un registro concreto
- Duplicado o copia de seguridad del fichero.
- Ordenación del fichero.
- Mezcla de los datos de varios ficheros para formar uno nuevo.

Muchas de las tareas de manipulación de los ficheros las realizará el sistema operativo y los programas que hagamos, normalmente realizarán peticiones al sistema operativo para gestionar los archivos

Tipo Puntero

En programación por **puntero** entendemos un tipo de dato que corresponde a una dirección de memoria que a su vez referencia a un dato de otro tipo.Una variable de tipo puntero lo único que va a contener por tanto es una dirección de memoria (una referencia) que será la que realmente contendrá el dato.

Cuando el compilador traduce a código máquina necesita darle al nombre de la variable una dirección de memoria si sabemos el tamaño se reserva esa zona de memoria. Esto es un **uso estático de la memoria**. Pero hay veces que no sabemos el tamaño. Para ello usamos los punteros, el compilador reserva un espacio de memoria para guardar la dirección de memoria de la variable, cuando el programa se ejecute buscaremos un trozo de memoria libre para almacenar el dato ya que es entonces cuando sabemos su tamaño y anotamos en nuestro puntero donde comienza la cadena (dirección de memoria). Con esto se consigue una **gestión dinámica de la memoria.**

En Java no existen punteros como tales, pero sí existen **referencias**, que conceptualmente son casi lo mismo, pero de manejo y gestión mucho más simple.

Tipo Lista

Un tipo de estructura dinámica son las ***Listas enlazadas.***

- Distintos elementos o nodos, cada uno de los cuales será un registro del mismo tipo, dispuestos de forma secuencial.
- En cada nodo se incluye un campo que sea de tipo "puntero al siguiente nodo de la lista"
- Un puntero de inicio que siempre apunte al primer elemento de la lista.
- El campo "puntero al siguiente nodo de la lista" del último nodo debe apuntar a un valor especial, que normalmente recibe el nombre de Null, y que indica que no hay ningún otro elemento detrás.

Tipo Arbol

Un **árbol** es una estructura de datos dinámica, que se construye usando punteros. La diferencia es que se trata de una estructura jerárquica.

- Un nodo llamado raíz del árbol por el que se accede a la estructura.
- Cada nodo puede tener varios nodos hijos, cada uno de los cuales se puede considerar como la raíz de un nuevo subárbol, y que constituye una rama del árbol general.
- Cada nodo deberá contener campos de tipo "puntero a nodo hijo" para cada uno de los posibles hijos.
- Para un nodo concreto, sólo existe un nodo padre, por lo que un nodo no puede pertenecer más que a una rama del árbol.
- Cuando una rama se termina, los campos que apuntan a los hijos, deben contener el valor Null.
- Es necesario disponer de un puntero que apunte siempre al nodo raíz del árbol, para poder acceder a la estructura, ya que si no sería imposible usarla, aunque estuviera en memoria ocupando espacio.

Datos definidos por el usuario

Algunos lenguajes permiten definir datos al usuario mediante distintos mecanismos. De ellos, los más útiles son los Objetos, en aquellos lenguajes que permiten programación orientada a objetos, ya que sí permiten definir auténticas estructuras de datos.

Enumerados

Son enumeraciones de una serie de datos específicos, por ejemplo los días de la semana. En Java existe algo parecido a través del interface **Collection**, y de toda una serie de clases que implementan ese interfaz

Subrango

Por ejemplo los días del mes es un subrango del rango de enteros.

Objetos

Los objetos tratan de permitir una libertad total al definir un tipo de dato:

- Nos permiten definir cualquier estructura de datos,

- cualquier conjunto de valores posibles,
- cualquier conjunto de operaciones que se pueden realizar con esos datos
- cualquier grado de libertad en la definición que permita adaptar los programas a las cosas del mundo real.

Está disponible para los lenguajes orientados a objetos. A través de la definición de una **clase**, se define una estructura de datos, tanto en lo relativo a qué valores va a tomar como en lo relativo a qué operaciones se van a poder aplicar a esos valores. Por otro lado, en la clase podemos definir todo un conjunto de **métodos** (o procedimientos) en los que se define qué uso se va a poder hacer de esos datos, junto a restricciones de acceso y seguridad.

Base de datos

Las **bases de datos** es una estructura que permite almacenar, consultar, actualizar y gestionar grandes cantidades de información, de forma cómoda y segura, y añadiendo funcionalidad extra al procesamiento que puede hacerse mediante ficheros individuales, ya que mediante un sistema gestor de bases de datos podremos realizar cómodamente consultas en las que se vean involucrados campos de múltiples registros que están contenidos en distintos ficheros, por ejemplo. De hecho, la base de datos estará formada por múltiples ficheros de distintos tipos, pero relacionados entre sí.

UNIDAD 3: Operadores

Trabajo de los compiladores

El proceso de un compilador viene a ser el siguiente:

- Cuando el compilador empieza a analizar el texto del programa escrito en lenguaje de alto nivel, lo que recibe es una secuencia de caracteres pertenecientes al alfabeto de ese lenguaje (Unicode, en el caso de Java, ASCII, en otros lenguajes)
- La primera tarea que debe realizar es aislar los elementos que tienen significado por sí mismos, lo que usando la analogía con el lenguaje natural serían las palabras que forman ese texto (ese programa, en nuestro caso). A esa fase se le llama **análisis léxico**.
- Para esto se sirve de unos elementos especiales que denominamos separadores. Un **separador** no es más que un carácter, un símbolo o una palabra escrita en el texto del programa que le indica al compilador dónde termina una palabra con significado propio para el lenguaje y dónde empieza la siguiente. Cada una de las "palabras" que reconoce el compilador recibe en programación el nombre de **token**.

Los tipos de separadores son los siguientes normalmente:

- **Espacio en blanco**. Es el separador típico
- **Return**. se usa con el mismo significado que el espacio en blanco. La única diferencia es la visual a la hora de leer el texto escrito.
- **Tabulador**. se trata de otro posible separador equivalente al espacio en blanco.
- **Comentario**. cumple asímismo las funciones de separador. Los comentarios tienen la función de documentación del código fuente

Tipos de token

- **Palabra reservada:** Son aquellos tokens que tienen asignada una función específica en el lenguaje y que los programadores no pueden utilizar más que para lo que el lenguaje establece.
- **Literales:** Son valores concretos para un tipo de datos básico del lenguaje. Por ejemplo, 3 es un literal para el tipo **int** en Java, y 3.0 es un literal para el tipo **double**, 'a' es un literal para el tipo **char**, y "Hola a todos" es un literal de tipo **String** (cadena de caracteres).
- **Operadores:** Son tokens especiales, porque actúan de separadores, manteniéndose a sí mismos como tokens, con significado propio. Los operadores permiten unir literales e identificadores (y también expresiones ya formadas) para formar expresiones más complejas

int numero = 38 donde **int** es una palabra reservada, **numero** es un identificador, **“=”** es un operador y “**38”** es un literal.

Expresiones

Los literales, junto con los identificadores y los operadores se pueden combinar para formar unidades de mayor significado, que denominamos expresiones. De esta forma la expresión podrá

evaluarse, (evaluar una expresión consiste en calcularla, sustituir cada elemento por su valor, y hacer las operaciones indicadas) dando como resultado un valor determinado. La expresión tendrá asociado un tipo, y ese tipo será el del valor resultante de su evaluación.

Las expresiones, combinadas con algunas palabras reservadas y a veces por sí mismas, forman **sentencias o instrucciones**.

- Todo literal de un determinado tipo es también una expresión de ese mismo tipo.
- Todo identificador de un determinado tipo (una variable o constante) es también una expresión de ese mismo tipo.
- Dado un operador *n*-ario que requiere *n* operandos (expresiones) de los tipos $t_1, t_2, ..., t_n$, y dados *n* operandos de los tipos correspondientes, el resultado es una expresión del tipo generado por el operador.

Expresiones matemáticas o aritméticas

Las expresiones matemáticas o aritméticas se forman mediante el uso de los operadores matemáticos o aritméticos asociados a los tipos numéricos básicos, junto a operadores de asignación. Generarán como resultado un valor numérico de alguno de los tipos definidos en el lenguaje.

Es posible que en una expresión matemática participen literales, identificadores (variables o constantes) o subexpresiones de distintos tipos numéricos, si bien es cierto que para operar entre ellos, habrá que hacer ciertas conversiones de tipo (también denominadas **castings**)

Las expresiones matemáticas deben dar como resultado de ser evaluadas, un valor numérico. Son los **operadores matemáticos**.

- Suma (+): Operador binario (necesita dos operandos)
- Número positivo o signo (+): Operador monario o unario (necesita un operando)
- Resta (-): Operador binario
- Número negativo o signo (-): Operador monario o unario
- Multiplicación (*): Operador binario
- División (/): Operador binario
- Resto-Módulo (%): Operador binario

El **operador de asignación** permite asignar un valor (que puede ser un literal o el resultado de evaluar una expresión) a un identificador (constante o variable).

Los operadores de incremento:

- El operador de incremento es ++ y se encarga de sumarle uno a la variable que acompañe.
- El operador de decremento es -- y se encarga de restarle uno a la variable que acompañe.

Ambos operadores pueden usarse en **notación prefija o postfija**. La diferencia es apreciable sólo cuando intervienen en una sentencia de asignación, ya que:

- En notación prefija (operador delante de la variable), primero se hace el incremento o

decremento y luego la asignación.

- En notación postfija (operador detrás de la variable) primero se hará la asignación con el valor que tuviera la variable, y luego se realiza el incremento.

Expresiones lógicas o booleanas

Las expresiones lógicas se obtienen por el uso de operadores de tipo lógico y relacional, dando como resultado un valor lógico o booleano.

Los **operadores relacionales** nos permiten formar expresiones lógicas por comparación de los valores usados como operandos. El resultado de estos operadores es un valor lógico (true o false, en Java). Para que puedan ser comparables, los valores deben corresponder a un tipo ordenado, comparable, tales como los números.

> Mayor que >= Mayor o igual que < Menor que < = Menor o igual que = = Igual != Distinto (no igual)

Los **operadores lógicos o condicionales** nos permiten formar expresiones lógicas directamente a partir de los valores de otras expresiones lógicas. Los tres operadores lógicos básicos son: **negación** (u operador **Not**), **conjunción** (o Y-lógico u operador **And**) y **disyunción** (u O-lógico u operador **Or**).

En el caso de la conjunción es posible que en una condición formada por operadores lógicos, no todos los componentes de la expresión se lleguen a evaluar. En una conjunción, si la primera parte es falsa, la segunda parte no se evalúa. En el caso de la disyunción es similar, si el operado primero es verdadero no se evalua la siguiente expresión.

Expresiones de bits

Las expresiones de bits, serán por tanto las que trabajan directamente con bits, usando operadores de bits.

- ˜ Negación (complemento de bits)
- & Conjunción binaria
- | Disyunción binaria (inclusiva)
- ˆ Disyunción binaria (exclusiva)
- << Desplazamiento a la izquierda, rellenando con ceros los bits que quedan libres a la derecha.
- >> Desplazamiento a la derecha, rellenando con el bit mayor (de signo) los bits que quedan libres a la izquierda.
- >>> Desplazamiento a la derecha, rellenando con ceros los bits que quedan libres a la izquierda.

Orden de precedencia de los operadores matemáticos

Cuando en una expresión intervienen varios operadores de distintos tipos, hay que dejar claro cuál va a ser el orden en el que se van a ejecutar. Esto es siempre posible mediante el uso de paréntesis, pero abusar de los paréntesis complica las expresiones hasta hacerlas a veces

ininteligibles. Aquí están en orden:

- Operadores postfijos expr++, expr--
- Operadores prefijos y unarios ++expr , --expr , +expr , -expr , ~ , !
- Creación y cast new , (tipo)expr
- Multiplicativos * , / , %
- Aditivos + , -
- Desplazamiento (bits) << , >> , >>>
- Relacionales < , <= ,> , >=
- Igualdad = = , !=
- Conjunción bits (AND) &
- Disy. excl. bits (XOR) ^
- Disy. incl. bits (OR) |
- Conjuncion lógica &&
- Disyunción lógica ||
- Condicional ? :
- Asignación += , -= ,+= , *= , /= , %= , &= , |= , ^= , <<= , >>= , >>>=

Regla de asociatividad de los operadores matemáticos y lógicos

Todos los operadores son asociativos por la izquierda, es decir, a igual nivel de precedencia y a falta de paréntesis, se realizarán en el mismo orden que están escritos, excepto la asignación, que es asociativa por la derecha. Resulve problemas como este: (2 / 4 / 8)

Funciones

Cualquier procedimiento o método que devuelve un dato, de cualquier tipo.

Sentencias

Las sentencias en un lenguaje de programación son el equivalente a las oraciones en el lenguaje natural. Es decir, las sentencias representan acciones completas a realizar por el programa. Todo programa no es más que un conjunto de sentencias o instrucciones. Debemos tener en cuenta:

- Sólo las expresiones de asignación forman sentencias por sí mismas, sin más que añadirles algún carácter o token de finalización de sentencia. Ese carácter en Java es el punto y coma.
- Todas las sentencias deberán terminar en punto y coma.
- Serán sentencias de asignación las obtenidas por el uso del operador de asignación básico o por cualquiera de los operadores combinados de asignación.
- También se consideran sentencias de asignación las obtenidas mediante los operadores de incremento o decremento, ya que llevan implícita una asignación.

Existen tres tipos básicos de sentencias:

- **Sentencias de expresión.** Están formadas por una expresión seguida del símbolo de terminación. Tipos:
    - **Expresiones de asignación**: expresiones cuyo operador raíz es bien el igual (=) o un igual compuesto (op=).
    - **Expresiones de incremento y decremento**
    - **Llamadas a métodos o procedimientos**
    - **Expresiones de creación de objetos**
- **Sentencias de declaración.** La declaración de una variable. Tipos:
    - Una sentencia de declaración puede declarar simultáneamente varias variables si son del mismo tipo
    - Una sentencia de declaración permite asignar además del tipo un valor inicial a una variable
    - Es posible mezclar la capacidad de inicialización de la sentencia de declaración con la posibilidad de declaración de múltiples variables

  La mayoría de los lenguajes modernos son fuertemente tipados, es decir que obligan a indicar siempre el tipo de todas las variables antes de usarlas (declararlas). Java es fuertemente tipado.
- **Sentencias de control de flujo.** Se encargan de "contener" a las otras sentencias del programa, de forma que se indique el orden en que se van a ejecutar esas sentencias, y bajo qué condiciones. Tipos:
    - Secuencial
    - Condicional
    - Cíclica

UNIDAD 4: Algoritmos

Concepto de algoritmo

Un algoritmo es:

- Una "fórmula" para resolver un problema.
- Un conjunto finito de acciones o secuencia de operaciones que ejecutadas en un determinado orden resuelven el problema.
- También puede definirse como un método para resolver un problema mediante una serie finita de pasos precisos y bien definidos.

El conjunto de todas las operaciones a realizar junto al orden en que deben efectuarse, se denomina **algoritmo**, y el lenguaje que permite representar esa solución de forma clara, y sin ambigüedades, es el **lenguaje algorítmico**. A la metodología necesaria para resolver problemas mediante programas se denomina **Metodología de la Programación**. El eje central de esta metodología es el concepto de algoritmo.

En el contexto de la informática, el algoritmo representa la secuencia de acciones o instrucciones que debe ejecutar el ordenador para solucionar un problema.

No es lo mismo saber resolver un problema que saber plantearlo de forma clara para que una máquina (ordenador) lo solucione.

Las características de un algoritmo son:

- Debe ser **preciso, sin ambigüedades**, **e indicar el orden** de realización de cada paso.
- Debe estar **bien definido**. Si se ejecuta dos veces dará los mismos resultados
- Debe ser **finito**.
- Es **independiente** tanto del lenguaje de programación en que se codifica como de la computadora que lo ejecuta.
- En programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras.
- En la mayoría de los casos **existen varios algoritmos para la solución** de un problema dado.
- La definición de un algoritmo suele incluir tres partes: Entrada, Proceso y Salida.

Clasificación de problemas

- **Problemas indecidibles**: aquellos que no se pueden resolver mediante un algoritmo.
- **Problemas decidibles**: aquellos que cuentan al menos con un algoritmo para su cómputo.
    - **Intratables**: aquellos para los que no es factible obtener su solución.
    - **Tratables**: aquellos para los que existe al menos un algoritmo capaz de resolverlo en un tiempo razonable.

Diseño y representación de algoritmos

Hay que tener en cuenta:

- Si es bastante complicado lo mejor es dividirlo en partes más pequeñas e intentar

resolverlas por separado. Esta metodología de "divide y vencerás" también se conoce con el nombre de diseño decendente

- Las ventajas de aplicar el diseño descendente son:
  - Al dividir el problema en módulos o partes se comprende más fácilmente.
  - Al hacer modificaciones es más fácil realizarlas sobre un módulo en particular que en todo el algoritmo.
  - Se probarán mucho mejor comprobando si cada módulo da el resultado correcto
- Una segunda filosofía a la hora de diseñar algoritmos es el refinamiento por pasos, que consiste en partir de una idea general e ir concretando cada vez más esa descripción hasta que tengamos algo tan concreto

Una vez solucionado hay que representarlo. Las representaciones más usadas son los **flujogramas** (o diagramas de flujo), los **diagramas NS** y el **pseudocódigo**.

Al escribir el algoritmo hay que tener en cuenta:

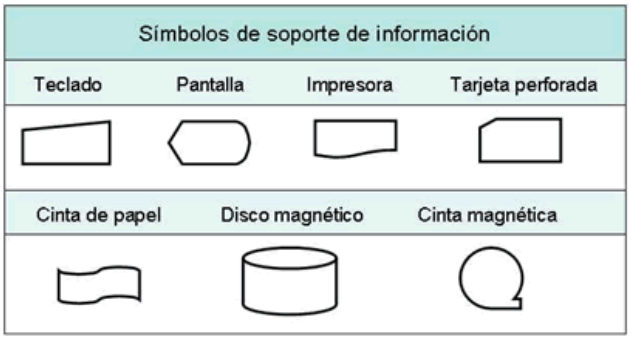
- En cada momento sólo se puede ejecutar una acción o **sentencia** (también llamada instrucción).
- Dentro de las sentencias del algoritmo pueden existir "palabras reservadas"
- Si estamos utilizando pseudocódigo tenemos también que usar la **indentación**

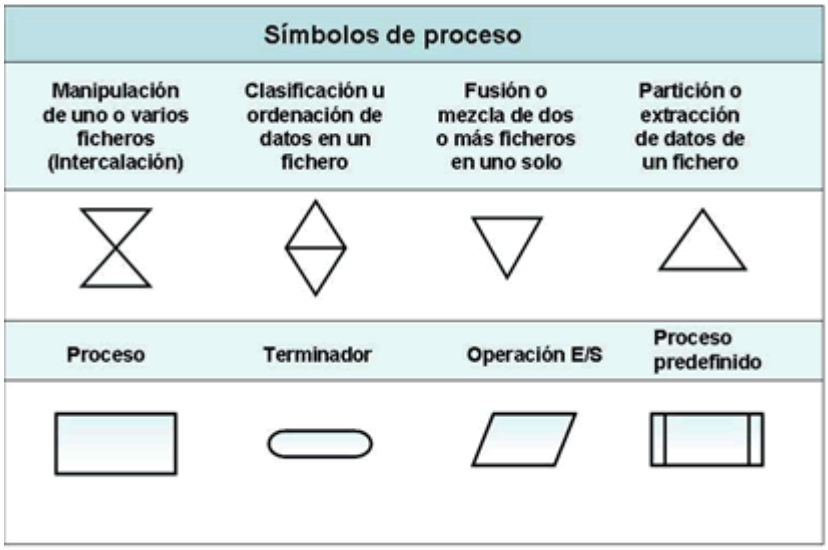
Diagramas de flujo

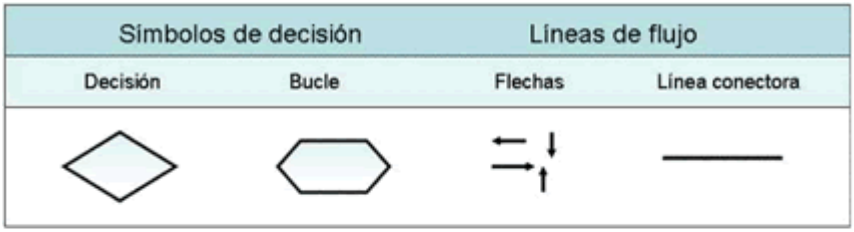
No son más que una herramienta gráfica, una notación gráfica estándar que posibilita la comprensión de los pasos y el orden en que hay que darlos para resolver un problema (algoritmo). Se basa en la utilización de unos símbolos gráficos que denominamos cajas, en las que escribimos las acciones que tiene que realizar el algoritmo. Deben ser leídos de arriba abajo, y de izquierda a derecha. Características:

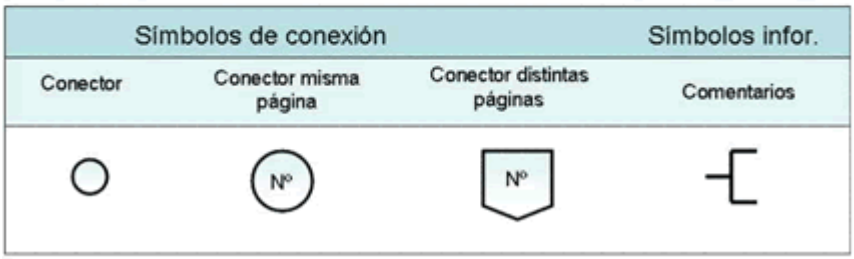
- Visual
- Adecuados para algoritmos no muy grandes
- Engorrosos de corregir

Las cajas están conectadas entre sí por líneas y eso nos indica el orden en el que tenemos que ejecutar las acciones. En todo algoritmo siempre habrá una caja de inicio y otra de fin, para el principio y final del algoritmo.





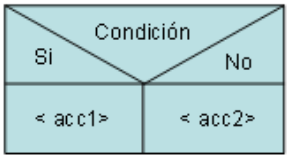




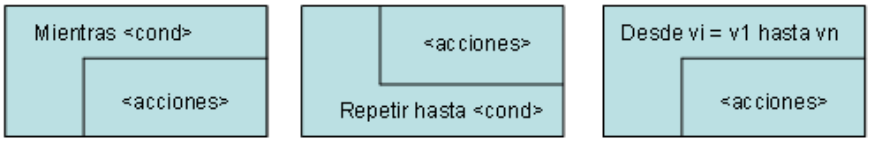
Diagramas N-S o de Nassi-Shneiderman

Son semejantes a los flujogramas, en el sentido de que también son una representación gráfica, pero sin flechas y cambiando algo los símbolos de condición y repetición. Las cajas van unidas. Pretenden presentar las acciones de forma más parecida a como se introducirán en el programa, al igual que se hace en pseudocódigo, pero sin perder la ventaja visual

Condiciones:



Estructuras repetitivas:



Pseudocódigo

Es una forma intermedia entre el lenguaje natural y el lenguaje de programación para la descripción de la solución de un problema. Características:

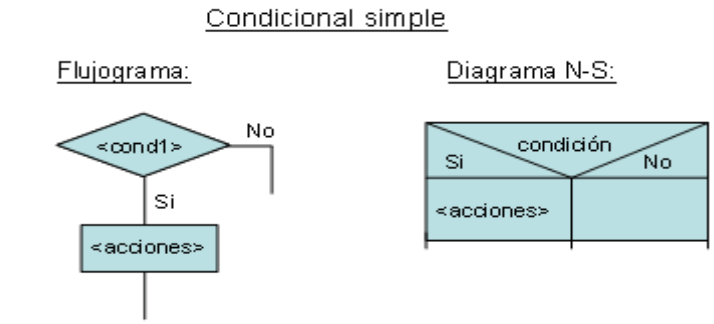
- Parecido a cualquier lenguaje de programación.
- Su traducción a cualquier lenguaje concreto es muy sencilla.
- No se rige por las normas de un lenguaje en particular, podemos hacerlo casi como queramos, siempre que lo que escribamos no sea ambiguo y sea claro.
- Se centra más en la lógica del problema que en los detalles de sintaxis.
- No es tan visual
- Son bastante más fáciles de hacer y de corregir que las representaciones gráficas, ya que son sólo texto.

Tiene tres partes: la **cabecera**, la **zona de declaración de constantes y variables** y el **cuerpo**.

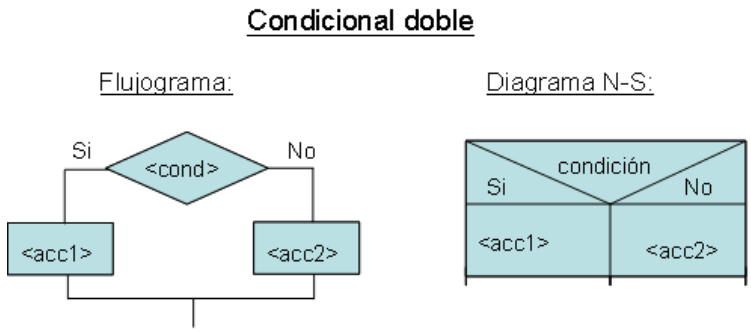
Estructuras básicas de control del flujo

Para todo problema con solución puede describirse un algoritmo que lo solucione usando sólo las tres estructuras de control de flujo básicas. Esas estructuras son las siguientes:

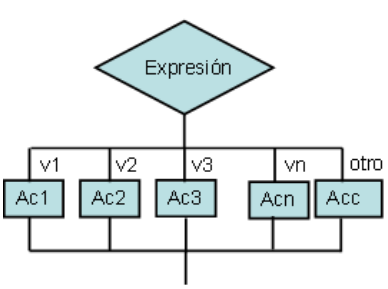
- **Secuencial:** es cuando una instrucción sigue a otra en secuencia, escritas en el orden en que van a ejecutarse.
- **Condicional o selectiva:** se evalúa la condición y en función del resultado se ejecuta un conjunto de instrucciones u otro. Existen:
  - Condicional simple.

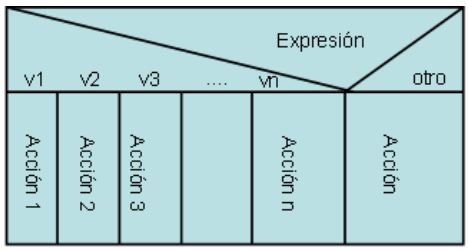


  - Condicional doble.



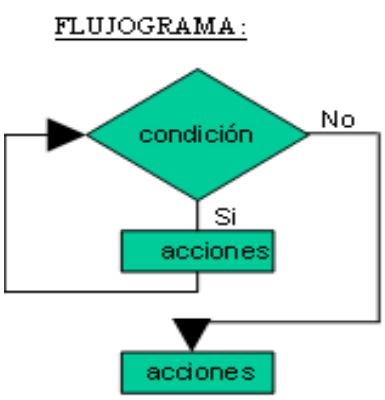
  - Selectiva múltiple



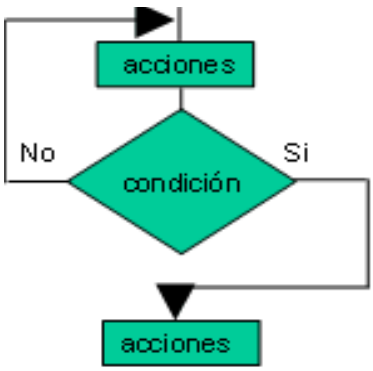


- **Cíclica, iterativa o repetitiva:** son aquellas que contienen un bucle (conjunto de instrucciones que se repiten un número finito de veces). Cada repetición del bucle se llama iteración.
    - Todo bucle tiene que llevar asociada una condición
    - Después de cada iteración se vuelve a evaluar la condición
    - Debe haber alguna o algunas instrucciones que modifiquen el valor de la condición

  Existen varias:
    - Tipo mientras-hacer (while-do): La condición del bucle se evalúa al principio, si es verdadera entramos en el bucle, al final volvemos analizar la condición hasta que sea falsa



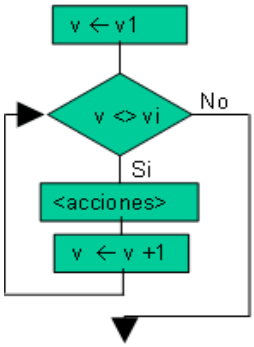
    - Tipo repetir-hasta (repeat-until): En este tipo de bucles, se repiten las sentencias que incluye hasta que la condición sea verdadera. La condición se evalúa al final, por lo que el bucle se ejecuta al menos una vez



- Tipo for: Este tipo de bucles se utiliza cuando se sabe ya antes de ejecutar el bucle el número exacto de veces que hay que ejecutarlo

ESTRUCTURA REPETITIVA FOR

FLUJOGRAMAS:



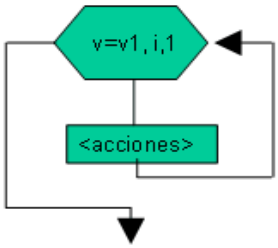
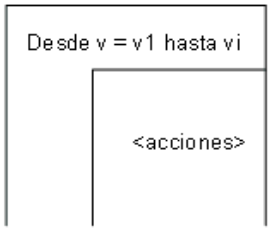


DIAGRAMA NS:



UNIDAD 5: Programación estructurada

Introducción

Se estima que hoy en día más del 80% del coste del software (de cualquier aplicación) no se debe al desarrollo propiamente dicho, sino a su mantenimiento (actualización, modificación, adaptación, extensión,...) . Tenemos que esforzarnos en aplicar técnicas que reduzcan el tiempo y coste.

Esas técnicas que permiten reducir los tiempos y costes de desarrollo y mantenimiento deberán ir encaminadas a conseguir programas **claros y fáciles de entender**, lo más **autodocumentados** posible y en los que se use **código reutilizable**.

Reglas para la “buena programación”

- **Diseño descendente**: Al diseñar un algoritmo partiremos de los conceptos y problemas más generales, que se irán descomponiendo en conceptos más simples, y continuaremos el proceso de "refinamiento paso a paso" hasta que lleguemos a conceptos y problemas de detalle, de fácil solución.
- **Diseño por módulos, o programación modular:** Dividiremos nuestro problema en partes independientes que puedan abordarse por separado, y solucionarse de forma independiente, incluso por personas o equipos distintos
- **Programación estructurada:** fija su atención en las estructuras de control de flujo que se pueden usar imponiendo algunas limitaciones y restricciones para evitar generar algoritmos difíciles de seguir y entender, y por tanto difíciles y costosos de mantener. Eso se consigue básicamente evitando **sentencias o estructuras de salto incondicional,** en las que en vez de ejecutar la siguiente sentencia o instrucción escrita en el programa, se salta a ejecutar otra sin que ello dependa de la comprobación de condición alguna. Sólo se permitirán por tanto estructuras de una entrada y una salida en el flujo del programa.
- **Relevancia de las estructuras de datos:** La elección de una forma adecuada de representar los datos en cada caso influye de forma decisiva en la utilización que puede hacerse de los mismos y en la eficiencia de los algoritmos y es importante seleccionar bien las estructuras de datos a usar.
- **Crear código reutilizable**

Reglas de la programación estructurada

- Las estructuras de control del flujo de un programa sólo deben tener un punto de entrada y un punto de salida
- No existen trozos de programa imposibles de ejecutar
- Beneficios
    - Mejora en la legibilidad y claridad del código resultante (programas más fáciles de comprender)
    - Mejora la productividad de los programadores

    - Disminuye los costes de la aplicación
- La ejecución de un programa estructurado progresa disciplinadamente, en vez de saltar de un sitio a otro de forma impredecible.
- Su fundamento teórico es el Teorema de Böhm-Jacopini (1966): "Cualquier programa de ordenador puede diseñarse e implementarse usando únicamente las tres construcciones estructuradas, que son secuenciación - selección - iteración. (Sin usar por tanto estructuras de tipo salto incondicional o tipo goto."
- Por tanto todo programa estructurado sólo usará esas tres estructuras de control de flujo (secuencia, selección e iteración)
- No existen bucles infinitos

Manejo de las estructuras de control de flujo en programación estructurada

- Estructuras condicionales o selectivas anidadas: los principios de la programación estructurada debe usarse indentación, la estructura interna tiene que estar totalmente dentro de la estructura externa y el if más interno tiene que estar totalmente dentro del if más externo.
- Ciclos o bucles anidados
- Terminación de un bucle preguntando antes de cada nueva iteración
- Terminación de un bucle indicando previamente el número de iteraciones a realizar: Bastará con llevar la cuenta de las que llevamos con un contador
- Terminación de un bucle usando un valor de salida: ponemos poner un valor especial, que usamos como centinela para saber cuando terminamos.
- Terminación de un bucle al agotar los datos de entrada: debemos disponer de algún mecanismo, o de alguna "marca" que nos permita preguntar si hemos alcanzado el final o no.

Estructuras que no corresponden con la programación estructurada

Este tipo de sentencias permiten saltar, sin comprobar ninguna condición a otra sentencia distinta de la que está escrita a continuación en el programa. Esa sentencia incluso puede ser cualquier otra del programa que hayamos etiquetado para poder saltar hasta ella.

- Usos admisibles de sentencias GOTO o de salto incondicional: se desaconseja el uso de esta estructura salvo contadas situaciones:
    - Salidas múltiples y "prematuras" de ciclos.
    - Transferir el control al código manipulador de errores en lenguajes que no disponen de un manejo apropiado de excepciones
    - Mejorar la eficiencia en programas con un tiempo de respuesta crítico.

  break (en java) es una sentencia que salta incondicionalmente al final de la estructura.
- Uso de Goto en salidas múltiples y "prematuras" de ciclos: usar goto cuando buscando en un vector por ejemplo
- Uso de Goto para mejorar la eficiencia en programas con un tiempo de respuesta crítico: Cuando se llama a un subprograma, todo el entorno volátil del programa en ejecución debe

guardarse en memoria, y asignarle espacio en memoria a todas las variables y objetos que utilice el subprograma. Una vez que finaliza el subprograma, debe recuperarse de la memoria el entorno volátil y restaurarlo en la CPU (para que el procesador sepa por donde dejó el trabajo), continuando el programa principal por el mismo sitio que lo dejó antes, pero teniendo a su disposición los datos que haya podido generar el subprograma ejecutado. Ese coste adicional en tiempo de procesamiento puede eliminarse usando Goto.

- Uso de Goto para manejo de excepciones: El programador debe tener previsto la posibilidad de que ocurran esos "accidentes", llamados excepciones, y en el caso de que se produzcan, detectarlos y tratarlos adecuadamente para evitar que el programa aborte.
- Uso inapropiado de estructuras de salto incondicional: sentencia return

Conclusiones

- La programación estructurada no pretende limitar la creatividad de los programadores, sino establecer **estándares** que faciliten su actividad y eviten el caos.
- Es deseable que todos los programadores de un proyecto sigan el mismo **estilo** de programación, para producir un código de calidad uniforme.
- El uso de proposiciones **Goto** debe **evitarse** en circunstancias normales.
- La profundidad de **anidamiento** de las construcciones de un programa no debe ser mayor de **cinco** en circunstancias normales.
- Los subprogramas no deben ser ni excesivamente pequeños, lo que aumentaría el coste de cada llamada en proporción al beneficio de no volver a escribir el código que contienen cada vez, ni demasiado grandes, lo que haría que fuesen difíciles de entender, y probablemente divisibles en más de un subprograma. Usualmente un **tamaño** entre 5 y 30 líneas de código es el apropiado.
- Apartarse de las circunstancias normales requiere un cuidadoso estudio de los pros y los contras, y normalmente deberá contar con la aprobación del líder o director del proyecto.

UNIDAD 6: Programación modular

Introducción

Es necesario programas más claros, más fáciles de mantener, menos costosos, más competitivos. ¿De qué manera hacía esto la programación modular?

- Dividiendo nuestro problema en partes independientes que puedan abordarse por separado.
- Solucionando cada parte de forma independiente, incluso por personas o equipos distintos.
- Permitiendo reutilizar módulos (trozos de código o programas) ya desarrollados (clases, herencia...).
- Independizando dentro de los módulos distintas partes y funciones distintas de nuestro problema (Diseño en tres capas: acceso a datos, negocio, presentación).
- Separando la definición de las estructuras de datos de los programas que las usan (definiendo tipos de datos abstractos)
- Independizando nuestra aplicación de las características físicas de los soportes sobre los que se almacenan los datos.
- Facilitando la depuración, las pruebas, la integración, el ajuste y la modificación de un sistema, que puede abordarse por partes, disminuyendo la complejidad.
- Aislando las dependencias funcionales

Concepto de módulo

Se pueden dar definiciones más o menos generales:

- Un módulo es una asignación de trabajo para un programador, que lo puede desarrollar de forma independiente.
- Un módulo es un conjunto de estructuras de datos y subprogramas que tienen cierta relación y coherencia entre sí (un paquete, en java).
- Un módulo es una subrutina, procedimiento, función o método que realiza una tarea concreta dentro de otro programa.

Un módulo no es una única cosa concreta, sino una "**abstracción mental**" que hacemos para organizar mejor la tarea de programación, y que luego plasmamos de distintas maneras en nuestra forma de construir los programas. Esas abstracciones pueden referirse:

- A los datos (independizar la definición de las estructuras de datos de las aplicaciones que las usan, definiendo esas estructuras dentro de módulos distintos)

Para definir un módulo se necesita:

- La estructura definirá que tipo de valores va a tomar (campos)
- Operaciones se podrán realizar (acciones: crear, modificar, etc...)
- Al control (independizar tareas distintas de nuestra aplicación dentro de módulos distintos).

La interface de un módulo consiste en definir claramente los datos de entrada que debe recibir de otros módulos y el valor que devolverá para que otros módulos puedan usarlo.

Características de los sistemas modulares

- Cada módulo se corresponde con un subsistema claramente definido, y que puede resultar útil para otras aplicaciones.
- La función de cada módulo tiene un propósito específico, definido claramente.
- Cada módulo maneja no más de una estructura de datos principal o relevante del sistema.
- Las funciones que manejan instancias de un tipo abstracto de datos se definen y quedan encapsuladas dentro de la estructura de datos de la que se trate.
- Las funciones de un módulo comparten datos globales de forma selectiva.

La modularidad de un sistema tiene grandes ventajas. Mejora la claridad del diseño de la aplicación, lo que facilita la codificación, la depuración, las pruebas, la documentación y el mantenimiento. De hecho, como parte de la documentación de cualquier aplicación suelen aparecer diagramas que representen los módulos de los que consta el sistema.

Criterios para definir la modularidad de un sistema

¿cuántos módulos debemos hacer?, ¿qué función tiene cada uno?, ¿qué debemos tener en cuenta? Ese tipo de preguntas deben ser contestadas en la fase de análisis de la aplicación, y darán como resultado la decisión de estructurar la aplicación en diversos módulos.

Objetivos al dividir el problema en módulos. Cada módulo será una **entidad bien definida** que tiene las siguientes características:

- Los módulos contienen tanto instrucciones o procesamiento lógico como estructuras de datos.
- Los módulos pueden ser compilados aparte y almacenados en una biblioteca o librería para poder ser reutilizados en otras aplicaciones.
- Habrá partes de un módulo que podrán utilizarse invocando un nombre con algunos parámetros (llamadas a métodos o subrutinas)
- Los módulos pueden usar a otros módulos.

Ejemplos de módulos son:

- Las subrutinas, métodos, procedimientos o funciones.
- Los grupos de métodos, subrutinas procedimientos o funciones que están relacionados por realizar una funcionalidad común.
- Las abstracciones de datos. (o tipos abstractos de datos)
- Las clases de programación orientada a objetos.
- Los paquetes que agrupan clases por funcionalidad o por pertenecer a la misma aplicación.

Existen muchos criterios de modulación y según el que usemos, el resultado será distinto.

- **Criterio convencional:** Cada módulo con sus submódulos corresponden a un paso en la secuencia de ejecución.
- **Criterio de ocultamiento de la información:** Cada módulo oculta a otros módulos una decisión difícil o modificable del diseño, de forma que pueda ser modificada sin que tengan que cambiar los demás módulos.
- **Criterio de abstracción de datos:** Cada módulo oculta los detalles de representación de una estructura de datos bajo un conjunto de funciones o métodos que acceden y modifican la estructura.
- **Criterio de acoplamiento y cohesión:** La estructura del sistema busca maximizar la cohesión interna de cada módulo y minimizar el acoplamiento entre módulos.
- **Criterio de limitar el tamaño máximo de cada módulo:** Cualquier módulo de un tamaño

grande seguramente incluye más de una función o más de una estructura de datos principal, susceptible de constituir un módulo independiente. Como regla general, no se aconsejan rutinas de más de 30 líneas de código ni módulos de más de 120.

Acoplamiento entre módulos

El acoplamiento entre dos módulos puede definirse como el grado de interdependencia entre ambos, es decir, en qué medida las operaciones que realiza uno pueden afectar a las que realiza otro de una forma clara o por el contrario como efectos colaterales difíciles de prever y controlar. También puede definirse como el número y complejidad de las interacciones entre distintos módulos.

- Mientras más acoplados estén dos módulos, más probable es que los cambios en uno obliguen a modificar el otro.
- Mientras más complejas sean las relaciones entre dos módulos, más difícil será darse cuenta de qué es lo que hay que modificar en los demás módulos al modificar uno con el que están fuertemente acoplados.

Buscaremos tener un sistema con módulos lo menos acoplados posible, y eso se consigue cuando la única interacción entre unos módulos y otros se produce a través de un interfaz bien definido, a través de los parámetros que se pasan en la llamada.

El módulo es una estructura de datos, su interfaz será la lista de métodos u operaciones que se pueden ejecutar con esa estructura. El máximo acoplamiento se dará cuando varios módulos comparten una misma zona de datos en memoria, a la que ambos acceden para hacer consultas y modificaciones.

Cohesión interna de cada módulo

La cohesión interna de un módulo es la fuerza con que están unidos los distintos elementos de ese módulo. Debe ser lo mas fuerte posible. Clasificación de más debil a más fuerte:

- **Cohesión coincidental:** Aparentemente los distintos elementos no tienen ninguna relación. Puede ser un programa de gran tamaño, que se ha segmentado en módulos arbitrariamente, o cuando se meten en el mismo módulo un conjunto de instrucciones que aparecen frecuentemente en otros módulos.
- **Cohesión lógica:** Distintos elementos se incluyen en el mismo módulo no porque se relacionen entre sí, sino porque realizan funciones similares. Las bibliotecas de funciones matemáticas pueden ser un ejemplo. La función "seno" no tiene más relación con la función "valor absoluto" de la que podría tener con cualquier otro método, procedimiento o función del problema, pero se agrupan en el mismo módulo sólo por ser funciones matemáticas.
- **Cohesión temporal:** Todos los elementos son ejecutados en el mismo momento, sin lógica alguna ni parámetro que determine cuál debe ejecutarse. Un ejemplo puede ser un módulo de inicialización, que asigna valores a un conjunto de variables o estructuras de datos.
- **Cohesión en la comunicación:** Cuando varios elementos no sólo se ejecutan en el mismo momento, sino que además se refieren al mismo conjunto de datos de entrada o salida.
- **Cohesión secuencial:** Cuando la salida de un elemento del módulo es usada como entrada del siguiente elemento del módulo.
- **Cohesión funcional:** Todos los elementos se encuentran relacionados por el desempeño

de una única función. Es un grado alto y por tanto deseable de cohesión.

- **Cohesión de la información:** Cuando el módulo contiene una estructura de datos compleja y varias rutinas o métodos que manejan esa estructura, presentando cada rutina cohesión funcional. Es la realización total de una abstracción de datos.

Otros criterios

- Ocultamiento de las decisiones complejas o modificables del diseño (lo importante de un módulo es lo que hace, no cómo lo hace).
- Limitar el tamaño físico de cada módulo (Cualquier rutina de más de 30-40 sentencias, seguramente realiza varias funciones susceptibles de estar divididas en más de un módulo).
- Estructurar el sistema para mejorar la claridad y facilitar las pruebas.
- Aislar rutinas que sean dependientes de la máquina (trozos programados en ensamblador).
- Facilitar la labor de modificación.
- Aumentar la eficiencia en la gestión de la memoria, en sistemas con memoria limitada.
- Aumentar la eficiencia en sistemas en tiempo real o con tiempo de respuesta crítico.

UNIDAD 7: Estructura básica del lenguaje java

Historia de Java

En 1991, la empresa Sun Microsystems financió un proyecto de investigación sobre el uso de los procesadores en los dispositivos electrónicos de consumo inteligentes. Se necesitaba un lenguaje de programación muy portable, capaz de correr en cualquier plataforma, debido a la diversidad de dispositivos electrónicos que podrían programarse. Uno de los resultados del proyecto fue la creación de un lenguaje de programación basado en C y C++, al que su creador (James Gosling) llamó Oak (roble en inglés) inspirado por un roble que crecía junto a la ventana de su trabajo en Sun. Como ya existía un lenguaje de programación que se llamaba así, hubo que cambiarle el nombre, y alguien propuso el nombre de Java (popularmente café en inglés) tras una reunión de un grupo de empleados de Sun en una cafetería. Y el nombre cobró fuerza. Nació el lenguaje Java.

Necesitaban desarrollar un lenguaje sencillo y capaz de generar código de tamaño muy reducido. Por tanto, los ingenieros de Sun desarrollaron un código "neutro", llamado bytecodes, que no depende del tipo de procesador, (del tipo de electrodoméstico), que se ejecuta sobre una "máquina hipotética o virtual" denominada Java Virtual Machine (JVM). Es la JVM quien interpreta el código neutro (bytecodes) convirtiéndolo a código particular de la CPU utilizada (código máquina). Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "Write Once, Run Everywhere". Podríamos traducirlo de forma más o menos libre como "escribe y compila una vez, ejecuta donde quieras".

Aunque el proyecto original estuvo a punto de cancelarse, debido al poco interés mostrado por los fabricantes de electrodomésticos, en 1993 la popularidad de Internet iba en aumento e hizo que Sun advirtiera la potencialidad de Java para la creación de páginas web con contenido dinámico, como consecuencia del modelo de traducción adoptado: el modelo de pseudocompilación a bytecodes. La clave fue, a finales de 1995, la incorporación de un intérprete Java en el navegador Netscape Navigator, versión 2.0, produciendo una verdadera revolución en Internet.

Java 1.1 apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje y posteriormente han aparecido constantes actualizaciones.

Java funcionaba en cualquier ordenador que estuviese conectado a la red, con independencia de su procesador o de su sistema operativo, a condición de que el navegador tenga instalada la máquina virtual Java (JVM- Java Virtual Machine). La JVM es el intérprete encargado de traducir los bytecodes al código máquina concreto de cada ordenador.

Características de Java

Al programar en Java no se parte de cero. Cualquier aplicación que se desarrolle se apoya en un gran número de clases que ya están disponibles. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el API o Application Programming Interface - Interfaz del Programador de Aplicaciones- de Java). Sus características son:

- **Simple**: Java es tan potente y funcional como cualquier otro lenguaje con el que se compare, pero sin las características menos usadas y más confusas de éstos. Elimina

muchas de las características de éstos, entre las que destacan: aritmética de puntero, registros (struct), definición de tipos (typedef), macros (#define) y necesidad de liberar memoria (free)

- **Orientado a Objetos:**
    - Sus datos son objetos e interfaces a esos objetos.
    - Soporta las 3 características de POO: encapsulación, herencia y polimorfismo.
    - Las plantillas de objetos son llamadas, como en C++, clases y sus copias, instancias.
    - Estas instancias, como en C++, necesitan ser construidas y destruidas en espacios de memoria.
    - Java incorpora funcionalidades inexistentes en C++ como por ejemplo, la resolución dinámica de métodos.
    - En C++ se suele trabajar con librerías dinámicas (DLLs) que obligan a recompilar la aplicación cuando se retocan las funciones que se encuentran en su interior. Este inconveniente es resuelto por Java mediante una interfaz específica llamada RTTI (RunTime Type Identification- Identificación de Tipos en Tiempo de Ejecución). Las clases en Java tienen una representación que permite a los programadores interrogar en tiempo de ejecución por el tipo de clase y asociar dinámicamente la clase con el resultado de la búsqueda.

- **Distribuido**: Java se ha construido con grandes capacidades de interconexión TCP/IP Existen librerías de rutinas para acceder e interactuar con protocolos como el protocolo http y el protocolo de ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales. La verdad es que Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas, interactuando.

- **Robusto**: Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución.
    - La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo.
    - Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error.
    - Maneja la memoria para eliminar las preocupaciones por parte del programador
    - Implementa los arrays auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobrescribir o corromper memoria resultado de punteros que señalan a zonas equivocadas.
    - Para asegurar el funcionamiento de la aplicación, realiza una verificación de los bytecodes.
- **Arquitectura neutral:** Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (run-time o máquina virtual Java) puede ejecutar ese código objeto
- **Seguro**: Integran punteros y casting implícito como hacen los compiladores de C y C++ para prevenir el acceso ilegal a la memoria. El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de bytecodes que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal
- **Portable**: Java construye sus interfaces de usuario a través de un sistema abstracto de

ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

- **Interpretado**: El intérprete Java (sistema run-time o Máquina Virtual Java) puede ejecutar directamente los bytecodes. No obstante, Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado antes de ser ejecutado en vez de ser ejecutado directamente como sucede en cualquier programa compilado de forma tradicional. En realidad java es tanto interpretado como compilado.
- **Multihebrado**: Java permite muchas actividades simultáneas en un programa. Los hilos (a veces llamados, procesos ligeros), son básicamente piezas independientes (líneas de ejecución) de un proceso. No ha de confundirse un hilo con un proceso. Hilo es la unidad mínima ejecutable, mientras que proceso es la unidad mínima planificable. Un proceso está formado por hilos. Por ello tiene mejor rendimiento interactivo y mejor comportamiento en tiempo real.
- **Dinámico**: Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Java también simplifica el uso de protocolos nuevos o actualizados.

Coceptos básicos sobre Java

- **JDK (Java Development Kit):** consiste, básicamente, en un compilador y un intérprete (JVM) para la línea de comandos. Actualmente recibe el nombre de J2SE Development Kit (JDK) (Java 2 Standard Edition Development Kit). Este Kit de desarrollo incorpora todas las librerías de la API (Aplication Programer's Interface - Interfaz para el Programador de Aplicaciones)

- **API de Java:** consiste en un juego de paquetes que se distribuyen con el JDK como bibliotecas de clases.

- **Variable de entorno PATH:** Tenemos que señalarle al sistema operativo dónde encontrar los ficheros que le indiquen cómo compilar y ejecutar ficheros Java. El desarrollo y ejecución de aplicaciones en Java exige que las herramientas para compilar (javac.exe) y ejecutar (java.exe) se encuentren accesibles. Si se desea compilar o ejecutar código en Java estos programas deberán encontrarse en el PATH. La carpeta BIN debe ser accesible e incluirse en el PATH
- **Variable de entorno CLASSPATH:** la variable de entorno CLASSPATH determina dónde buscar tanto las clases o librerías de Java (el API de Java) como otras clases de usuario. A partir de la versión 1.1.4 del JDK no es necesario indicar esta variable, salvo que se desee añadir conjuntos de clases de usuario que no vengan con dicho JDK. Por defecto, las clases se buscarán cuando se necesiten en el directorio de trabajo activo y en el directorio lib del JDK. La variable CLASSPATH puede incluir la ruta de directorios o ficheros comprimidos "*.zip" o "*.jar" en los que se encuentren los ficheros "*.class"

Niveles léxico, gramatical y semántico de los lenguajes de programación.

Las similitudes con el lenguaje natural podrían resumirse así:

- **Estructura lingüística**: El lenguaje de programación imita la estructura lingüística de los lenguajes, lo que supone la aparición de **los niveles léxico, gramatical y semántico.**
- **Implementación ejecutable:** Necesitamos algo que funcione en un ordenador, que sea

"ejecutable". Para ello existen normas que definen

- qué símbolos son correctos en el lenguaje,
- qué combinaciones de dichos símbolos forman palabras correctas en el lenguaje (nivel léxico),
- qué combinaciones de palabras forman sentencias (nivel gramatical) y
- qué combinaciones de sentencias tienen sentido y constituyen un programa correcto en el lenguaje (nivel semántico).

El alfabeto de Java: Unicode.

El lenguaje se construye sobre un alfabeto, un conjunto de símbolos que son los que podemos emplear para escribir nuestros programas. Tradicionalmente, el alfabeto que venía usándose para todos los lenguajes de programación era el código ASCII de 8 bits, lo que permitía un conjunto de 256 símbolos distintos, que más o menos coincidían con los caracteres usados en la mayoría de los países occidentales. Pero eso imposibilitaba el uso de caracteres propios de los idiomas de muchos países. Java ha incorporado como alfabeto el código Unicode, que es un nuevo código estándar de E/S que usa 16 bits para representar cada carácter, lo que da un número suficientemente amplio (2^{16} = 65.536 posibles caracteres distintos) . Además el código Unicode es totalmente compatible con ASCII, ya que respeta el código de los caracteres ASCII, a los que sencillamente les coloca 8 bits a cero delante para convertirlos en Unicode.

Nivel léxico: Tokenización.

Inicialmente el compilador percibe nuestro fichero de texto con el código fuente en Java como una secuencia de caracteres, que le van siendo suministrados uno a uno (tal y como se escriben) y de los que tiene que ir extrayendo las palabras con significado propio (los tokens, en la jerga informática referida a compiladores). Para separar los tokens se utilizan separadors. En Java, son separadores los siguientes caracteres o grupos de caracteres:

- Espacio en blanco
- Return: tiene las mismas propiedades que el espacio en blanco.
- Tabulador: podemos decir lo mismo, se trata de otro posible separador.
- Comentario: Un comentario (de cualquiera de los tres tipos posibles en Java) cumple asimismo las funciones de separador. Los comentarios poseen asimismo la función de documentación sobre el código fuente, pero desde el punto de vista léxico del lenguaje, no son más que separadores.
- Operador. Son un caso especial, porque actúan como separadores que permiten diferenciar tokens distintos, pero con la peculiaridad de que se mantienen a sí mismos como tokens significativos, por lo que no podrán ser sustituidos por ningún otro separador.

Tipos de token

Lo primero será determinar el tipo de esos tokens. Son:

- **Palabras reservadas:**
- **Operadores**
- **Literales:** sería el valor de una variable
- **Identificadores:** sería la variable en sí

Tipos básicos o primitivos.

- **Boolean:** Permite representar valores lógicos; Verdadero (**true**) o Falso (**false**).
- **Char:** Permite representar cualquier símbolo o carácter UNICODE de 16 bits.
- **Byte:** Entero de **8 bits** con signo (representado en complemento a dos). Su rango de valores va desde **-128 (-2^7)** a **+127 (+2^7-1)**.
- **short:** Entero de **16 bits** con signo (complemento a dos). Rango de valores entre **-32.768 (-2^{15})** y **+32.767 (+2^{15}-1)**.
- **int:** Entero de **32 bits** con signo (complemento a dos). Rango de valores entre **-2.147.483.648 (-2^{31})** y **+2.147.483.647 (+2^{31}-1)**.
- **long:** Entero de **64 bits** con signo (complemento a dos). Rango de valores entre **-2^{63}**y **+2^{63}-1**.
- **float:** Número real (en coma flotante) de 32 bits, utilizando la representación IEEE 745-1985.
- **double:** Número real (en coma flotante) de 64 bits, utilizando la representación IEEE 745-1985.

Literales de los tipos primitivos.

Cada tipo tiene por tanto su conjunto finito de valores válidos.

- **El tipo boolean:** true o false
- **Los literales del tipo char:** Se representan mediante comillas simples. Cualquier símbolo Unicode situado entre comillas simples es un literal de tipo carácter.
- **Literales de tipo entero.** Un número entero se define como una secuencia de dígitos (0-9) que puede llevar delante signo o no.
- **Literales de tipo real.** En Java hay dos tipos de números reales, que son float y double. La diferencia básica es la precisión de la representación y el rango de valores representables, debido al mayor tamaño de double, que usa 64 bits en vez de los 32 de float para representar cada valor. Un número se considera real si posee decimales (lo que supone que se utilice el punto de separación entre la parte entera y la decimal, aunque cualquiera de las dos puede estar vacía), o posee un exponente (el exponente se introduce mediante la letra e ó E), o va precedido por una letra identificativa de tipo (que es f ó F para float y d ó D para double).

UNIDAD 8: Estructura básica del lenguaje java (parte II)

Literales en Java

Los literales son valores concretos para un tipo de datos básico del lenguaje.

- **Tipo booleanos:** Representa datos de tipo lógico verdadero o falso.
- **Tipo char:** los literales de tipo char son cada uno de los caracteres del alfabeto que usa el lenguaje Java. Recordamos que el alfabeto usado por Java es el alfabeto Unicode, que gracias a los 16 bits que usa para representar cada carácter, permite representar una amplísima variedad de símbolos o caracteres (en total 2^{16} = 65.536 símbolos distintos). Los literales de caracteres se representan mediante comillas simples. No obstante, existen muchos caracteres Unicode, y no todos los podemos escribir directamente con nuestros teclados de 102 teclas. El lenguaje Java proporciona varias formas alternativas de representar los literales de tipo char.
    - Mediante secuencias de escape, usando la contrabarra (\) seguida de algún carácter que debe interpretarse de forma especial
        - '**\b**' Espacio hacia atrás '**\u0008**'
        - '**\n**' Nueva línea '**\u000A**'
        - '**\t**' Tabulador '**\u0009**'
        - '**\"**' La comilla doble '**\u0022**'
        - '\'' La comilla simple '**\u0027**'
        - '****' La contrabarra **\u005C**'
    - Mediante su código en hexadecimal al que le añadimos delante el carácter de escape \u (de unicode)
    - Podemos representar los caracteres ASCII mediante su código en octal tras el carácter de escape \
- **Tipo entero:** Un literal para un tipo entero se forma como una secuencia de dígitos (0-9). Los números enteros se pueden especificar en tres notaciones: decimal, octal y hexadecimal. Se escriben así:
    - Por defecto, todos los números se consideran que están escritos en base decimal
    - Los números que comienzan por un 0 (cero), se consideran que están en base octal
    - Los números que comienzan por 0x o 0X se tratan como números en hexadecimal

    Hay diferentes tipos de enteros:

    - byte 8 =1 byte 2^8 = 256 (-128 +127)
    - short 16 =2 bytes 2^{16} = 65.536 (-32.768 +32.767)
    - int 32 = 4 bytes 2^{32} = 4.294.967.296 (-2.147.483.648 +2.147.483.647)
    - long 64 = 8 bytes 2^{64} = 1,844674407 E+19 (-9.223.372.036.854.775.808 9.223.372.036.854.775.807)
- **Tipos reales:** números con cifras decimales

- float 32 = 4 bytes 2^{32} = 4.294.967.296 (1.401298464324817 a E-45 3.4028234663852886 E+38)
- double 64 = 8 bytes 2^{64} = 1,844674407 E+19 (4.9 E-324 a 1.7976931348623157 E+308)

Un número se considera de tipo real si cumple:

- Si posee decimales (se utiliza el punto como separación entre la parte entera y la decimal). Tanto la parte entera como la parte decimal puede estar vacía. Así 124. equivale a 124.0 y .85 equivale a 0.85
- Si posee un exponente (el exponente se introduce mediante la letra e ó E)
- Si va seguido por una letra identificativa de tipo real
  - f ó F para float 28f , 28F y 28.0f son literales de tipo float.
  - d ó D para double 28d , 28D , 28.0d y 28.0 son literales de tipo double.

En caso de no especificar la letra correspondiente al tipo float, todos los números reales se consideran en Java de tipo double por defecto.

Con respecto a los operadores, el operador \ es la división entera (sin sacar decimales) si los dos operandos son números enteros y será la división real (sacando decimales) si alguno de los operandos es de tipo real.

Identificadores en Java

Los identificadores como los nombres que el programador decide asignar a los elementos que crea en su programa. Un identificador es una secuencia de uno o más símbolos Unicode que cumple las siguientes condiciones:

- Puede tener cualquier longitud, no hay tamaño máximo.
- El primer símbolo de la secuencia es una letra, un símbolo de subrayado (_) o un símbolo dólar ($).
- El resto de caracteres de la secuencia pueden ser letras o dígitos mezclados indistintamente.

Se podrá:

- Se puede usar cualquier símbolo Unicode,
- No hay límite de tamaño para un identificador, pero la lógica impone usar nombres no excesivamente largos,
- No se pueden usar espacios en blanco en medio de un identificador.
- Aunque no es obligado su cumplimiento, debe seguirse el convenio para nombrar identificadores en Java:
  - Variables, constantes de objeto, métodos, etc: La primera palabra del identificador en minúscula, y sólo la primera letra de cada una de las siguientes palabras en Mayúsculas. Ej: nombreDelEmpleado
  - Clases e interfaces: Igual, pero la primera letra de la primera palabra, también en mayúscula. Ej: EmpleadoEmpresa
  - Constantes de clase:Todas las letras en mayúsculas, y las palabras separadas por el símbolo de subrayado. Ej: MAXIMO_VALOR_PERMITIDO
- Aunque existen caracteres no permitidos, lo mejor es usar nombres razonables, y si el

compilador en algún momento nos indica que hay un error en el identificador, lo corregimos.

- Las palabras reservadas no pueden usarse como identificadores.

Operadores en Java

Los identificadores como los nombres que el programador decide asignar a los elementos que crea en su pro

- **Operadores aritméticos:**
  - Suma + Operador binario (necesita dos operandos)
  - Número positivo o signo + Operador monario o unario (necesita un operando)
  - Resta - Operador binario
  - Número negativo o signo - Operador monario o unario
  - Multiplicación * Operador binario
  - División / Operador binario
  - Resto-Módulo % Operador binario
- **Operadores relacionales:**
  - > Mayor que
  - >= Mayor o igual que
  - < Menor que
  - <= Menor o igual que
  - = = Igual
  - != Distinto (no igual)
- **Operadores lógicos:**
  - ! Negación (not)
  - && Conjunción (and)
  - || Disyunción (or)
- **Asignación** = , += , -= , *= , /= , %= , &= , |= , ^= , <<= , >>= , >>>=
- **Operadores de incremento o decremento:** expr++ , expr-- , ++expr , --expr
- **Operadores de bit:**
  - Negación de bits ~
  - Desplazamiento (bits) << , >> , >>>
  - Conjunción bits (AND) &
  - Disy. excl. bits (XOR) ^
  - Disy. incl. bits (OR) |
- **Operador new de creación de objetos** (new)
- **Operador de casting o conversión explícita de tipos:** El operador de casting explícito, consiste en poner un tipo entre paréntesis delante de alguna variable, constante, objeto, literal o expresión que sea de otro tipo. (tipo)expr

- **Operador Condicional** (? :). El operador condicional no es más que una versión abreviada de una sentencia tipo if -then-else. Su uso es:

  <condición> ? <Expresión si condición verdadera> : <Expresión si condición falsa>

- **Asociatividad y Precedencia de operadores.** Todos los operadores son asociativos por la izquierda, es decir, a igual nivel de precedencia y a falta de paréntesis, se realizarán en el mismo orden que están escritos, excepto la asignación, que es asociativa por la derecha.

UNIDAD 9: Sentencias y control de ejecución en java

Tipos de sentencias en Java

- **Sentencias de expresión.** Están formadas por una expresión seguida del carácter; (punto y coma). Ej: Asignación, incremento y decremento, llamadas a métodos y creación de objetos. =, +=, -=, *=, /=, %=, ++ y --
- **Sentencias de declaración**. La declaración de una variable, en la que se le asocia al identificador de la variable un tipo, y opcionalmente un valor, que genera una sentencia, al añadirle el símbolo de terminación. El compilador de Java siempre que encuentra un tipo seguido de un identificador, interpreta que se trata de una declaración..Java es un lenguaje fuertemente tipado. Algunas veces en la declaración aparecen otro palabras, como “final” que indica que la variable es una contante y cuando se le asigne un valor no podrá ser modificado. Otra palabra como “private” indica que sólo será posible modificarse dentro de la mism clase. Otra como “static” indica que es una variable de la clase y no del objeto. Estas palabras se llaman **modificadores**. Ej: declaración del tipo de variables, declaración e inicialización de variables
- **Sentencias de control de flujo.** Se encargan de "contener" a las otras sentencias del programa, de forma que se indique el orden en que se van a ejecutar esas sentencias, y bajo qué condiciones. Pueden considerarse como sentencias estructurales dentro del programa.
    - Ejecutación secuencial:
        - En Java todas las sentencias se terminan con un carácter de punto y coma (;)
        - Existe la sentencia nula, que es el carácter punto y coma.
        - Se pueden definir bloques de sentencias.
    - Ejecución condicinal: se usa if-else. También podemos usar if múltiple y condiciones anidadas
    - Ejecución cíclica: setencias “while” o “do - while”
    - Condicional múltiple o múltiple selectiva: usan “switch” y se caracteriza:
        - debe ser de tipo entero, es decir, debe devolver un valor de tipo entero.
        - Los tipos char, byte y short se convierten automáticamente a **int** (enteros) al operar sobre ellos
        - Cada caminos vendrá especificado por una cláusula **case**
        - las cláusulas case no pueden indicar condiciones, ni rangos de valores, ni listas de valores.
        - Es posible indicar un caso por defecto (**default**)
        - Se produce una ejecución denominada por caída hacia abajo. la ejecución continúa por los siguientes case hasta que se encuentre una sentencia de interrupción: break;
    - Setencias de saltos incondicionales: La sentencia “break” transfiere el control al final del ciclo “while”, “do while”, “for”. La sentencia “continue” lo que hace es devolver el control al bucle que se interrumpe, y que será el más interno, el que contiene la sentencia. La sentencia “return” se usa con una doble finalidad**:** Termina

la ejecución del método en el que se encuentre, transfiriendo el control al punto desde el que se hizo la llamada a ese método, continuando con la sentencia posterior y si va acompañado de una expresión de un determinado tipo, hace que el método devuelva un valor

UNIDAD 10: Recursividad

Concepto de recursividad

Usamos el término recursividad para referirnos a procedimientos o métodos que contienen en su implementación llamadas a sí mismo, es decir un método es recursivo si para ejecutarse se llama a sí mismo.

¿Qué debemos tener en cuenta a la hora de saber si un método recursivo está bien definido?

- Además de la llamada al propio método, deben existir otras sentencias en la definición del método que establezcan al menos un caso base, es decir, un caso para el que se conoce la solución, y para el que la llamada al método puede devolver un valor o terminar la ejecución sin necesidad de volver a invocar de nuevo al propio método.
- Cada nueva llamada al propio método debe reducir el problema, es decir, debe acercarnos más al caso base.
- La solución debe ser correcta para los casos no base. Es decir, debemos ser capaces de expresar correctamente la solución de un caso a partir de la soluciones de casos menos complejos, para poder construir la solución de cualquier caso a partir de las soluciones del caso base.

La recursividad es una poderosa herramienta en programación y ofrece una alternativa a las soluciones iterativas complejas de algunos algoritmos. ¿Cuáles son entonces las razones para usar la recursividad?

1. Los problemas resultan más fáciles de resolver que con estructuras iterativas.
2. Proporciona soluciones más simples.
3. Proporciona soluciones elegantes.

Siempre existe una solución iterativa, y ésta siempre será más eficiente, es decir, necesitará consumir menos recursos de memoria y de CPU, y obtendrá la solución en menos tiempo.

Lo que ocurre es que a veces, por la naturaleza recursiva del problema, no es fácil encontrar la solución iterativa, o resulta complicada y difícil de entender. La recursividad se debe usar cuando sea realmente necesaria, es decir, cuando no exista una solución iterativa simple. Cuando las dos soluciones son fácilmente expresables, siempre será preferible la solución iterativa.

Por otro lado, la recursividad requiere hacer una **asignación dinámica de memoria** (se reserva memoria cuando se necesite para almacenar un valor de un tipo dado. Después, una vez que no se necesite el valor, es posible liberar la memoria y hacerla disponible para otro uso por el sistema), que no es posible en todos los lenguajes de programación. (Sí lo es en Java y en otros muchos).

Cada vez que se hace una nueva llamada al **método recursivo**, es necesario guardar en la **pila (stack)** todos los datos de la llamada anterior que aún no ha terminado (todas las variables de memoria que tenga definidas el método), lo que se conoce como **entorno volátil** de esa ejecución. Esto incluye almacenar en la memoria todos los valores de los registros del **microprocesador**, para poder restablecerlos posteriormente y continuar por el mismo punto de la

ejecución de esa llamada que lo habíamos dejado. Como puede haber muchas llamadas sucesivas, éstas se van almacenando en la pila, de forma que se irán retirando de la pila en el orden contrario a como se introdujeron. El último en entrar es el primero en salir. (Estructura LIFO: Last-IN, First-Out)

Todo ese tiempo de CPU dedicado a:

- guardar los datos de cada ejecución en la pila,
- cargar los de la nueva llamada para cuando al final ésta termine
- ir de nuevo sacando esos valores de la pila en orden contrario a como se introdujeron
- para restaurarlos en la CPU y completar la llamada,

es la causa responsable de que las soluciones recursivas requieran mucho más tiempo de cpu y mucha más memoria.

Con una solución **iterativa** sólo tendremos un método ejecutándose, por lo que sólo necesitaremos memoria para una copia de cada una de las variables que usa ese método.

UNIDAD 11: Estructura estática de datos

Una estructura de datos se considera **estática** cuando al crearla,

- se le asigna un tamaño,
- una cantidad de memoria para su almacenamiento
- y no puede crecer o disminuir de tamaño según las necesidades del programa una vez que se está ejecutando.

Una estructura de datos **dinámica**, por el contrario,

- no se crea con un tamaño determinado, sino que ocupa siempre exactamente el tamaño que necesita para los elementos o datos que contiene.
- Si se elimina durante la ejecución del programa un elemento de la estructura, disminuye el tamaño necesario para el almacenamiento de la estructura, y se libera el espacio que ya no es necesario.
- Si se inserta un nuevo elemento, se busca espacio libre en memoria para ese nuevo elemento, y la estructura aumenta de tamaño, ocupando el que necesita.

Conceptos previos: modificador static

Las variables que definimos en un método son accesibles desde los otros métodos pero puedo definir una variable como **static**, consiguiendo que sea accesible desde cualquier parte del código de la clase en la que se ha definido. Incluso puedo consultar su valor y utilizarla desde otras clases, refiriéndome a ella a través del nombre de la clase en la que se ha definido. Es una **variable global** para la clase.

También se puede usar la palabra static junto a un método. Un método static no se ejecuta para un objeto, sino para la clase entera, de forma que podremos ejecutarlo aunque no hayamos creado ningún objeto de esa clase.

Conceptos previos: llamadas a métodos

Un método no es más que un trozo de código al que se le da un nombre, y que puede ser ejecutado invocando todo ese código mediante su nombre desde cualquier parte del programa. El resultado de la ejecución de cada llamada no va a ser siempre el mismo, sino que va a depender de los parámetros que le hayamos suministrado y ese resultado, puede ser la devolución de un valor, del tipo que sea, de forma que además de ejecutar las sentencias, la llamada puede incluirse en cualquier expresión, y será sustituida por el valor devuelto tras la ejecución del método, como si fuera una función en medio de una expresión matemática. La diferencia con las funciones, es que los métodos no sólo van a devolver datos numéricos, sino que pueden devolver datos de cualquier tipo definido en el programa. Cuando un método termina de ejecutarse, todas las variables locales que usa o que haya declarado, se borran de la memoria.

Los parámetros que se pasan en la llamada son variables locales al método, en las que se copian los valores pasados en la llamada. Sin embargo, con los objetos la cosa cambia. Si paso un objeto como parámetro lo que realmente estoy pasando es la referencia. En el parámetro local se hace una copia del valor de la referencia. Pero ese valor no es más que la dirección de memoria en la

que está el objeto real, por lo que el método trabajará con el objeto real.

En la definición de cualquier método siempre hay que indicar el tipo que devuelve, si no deseo un tipo devuelto debo indicar que el método devuelve el tipo "vacío". Esto se indica con la palabra reservada void. Los métodos constructores son los únicos para los que no hay que especificar ningún tipo devuelto, ni siquiera void.

Los parámetros son valores o variables que se indican entre paréntesis en la definición y en la llamada a un método. Los parámetros se utilizan en el cuerpo del método para llevar a cabo las tareas programadas en el mismo, e influyen directamente en el resultado de ejecutar dicho método, por ejemplo en el valor devuelto. Existen dos formas de pasar variables como parámetros a un método:

- Paso de parámetros por valor. Al método se le pasa el valor de la variable (una especie de copia)
- Paso de parámetros por referencia. Al método se le pasa la dirección de memoria donde está el objeto original (una copia de la referencia o dirección de memoria donde se guarda el objeto), por lo que se trabaja con el objeto en sí

Cadenas de caracteres

Java ha creado varias clases específicas para trabajar con cadenas de caracteres, y que son en cierta medida peculiares. La primera de ellas es la clase String, y la otra es la clase StringBuffer. A partir de la versión 5 del JDK, se incluye una clase **StringBuilder**, que es en todo similar a **StringBuffer**, pero que no permite sincronización entre procesos, por lo que resulta más eficiente para la gran mayoría de aplicaciones, que no requieren de esta característica.

Que todos los tipos primitivos tienen un tamaño fijo, conocido de antemano. En los tipos por referencia no podemos saber el tamaño exacto que va a ocupar cada objeto de una clase. Dos alternativas para reservar espacios en memoria:

- Reservar para cualquier variable de tipo String un mismo tamaño fijo: en la mayoría de los casos estaríamos desperdiciando mucha memoria
- Usar referencias. Es la alternativa usada en Java. Consiste en que no se reserva ningún espacio para las variables que no sean tipos básicos (llamadas variables por referencia, entre ellas las declaradas como String).

La existencia del **recolector automático de basura** permite evitar el uso de punteros que tanto complica la programación en otros lenguajes. Aunque las referencias son conceptualmente muy similares a los punteros, su uso y la gestión de estructuras de datos que las utilizan, es enormemente más sencillo que en el caso de los punteros. Ésa es una de las características de Java que hace que sea un lenguaje simple, y que reduce los tiempos de desarrollo y los errores en el código. A cambio de esa facilidad, se puede perder algo de eficiencia, ya que puedes tener en algunos momentos mucha memoria ocupada por basura inútilmente, a la espera de que se active el recolector de basura.

La clase String

String son variables por referencia, y que nos permiten representar cadenas de caracteres. Necesita invocar al operador **new** para crear nuevos objetos de esa clase. Sin embargo, la clase **String** se usa tanto, que los diseñadores del lenguaje incluyeron una versión abreviada de esa llamada al constructor.

- Como un tipo primitivo. Esto es, directamente asignándole un literal de tipo String.

  String nombre="Pepe";

- Invocando a alguno de los constructores de la clase. Esto es, usando el operador new.

  String nombre = new String("Pepe");

Otra peculiaridad de la clase String es que genera objetos inmutables, es decir, que no se pueden modificar desde que se crean hasta que se destruyen. Aparentemente sí se puede modificar el valor pero lo que ocurre en realidad es que cada vez que hacemos alguna modificación sobre el objeto String, se crea un objeto nuevo sobre el anterior modificado, y se actualiza la referencia.

La clase StringBuffer

La clase **StringBuffer** también nos permite representar cadenas de caracteres. De hecho la forma habitual de construir un objeto de tipo **StrinBuffer** es a partir de un objeto **String**.

```
String textoInicial = "Vamos a escribir un texto para procesarlo:"
StringBuffer textoDocumento = new StringBuffer (textoInicial);
```

Si en nuestro programa usamos numerosos objetos String, y sobre todo, si los modificamos con frecuencia, nos encontraremos con que el tiempo dedicado a crear el nuevo objeto modificado y a actualizar la referencia puede no ser despreciable e incidir negativamente en el rendimiento de nuestra aplicación. **StringBuffer define objetos que sí pueden ser directamente modificados**, sin necesidad de crear un nuevo objeto, ni de actualizar las referencias.

StringBuffer **reserva para cada cadena de caracteres un espacio extra de almacenamiento**, además del estrictamente necesario. Usa ese espacio extra para modificar el texto, si es más grande realojaría el objeto en otra zona de memoria.

Si para alguna variable de tipo cadena de caracteres sabemos que se van a producir **muchas operaciones** que la van a modificar, y que van a afectar a su tamaño, **debemos declararla de tipo StringBuffer**, de forma que **ganaremos en velocidad de procesamiento de esas modificaciones**, al no ser necesario realojar un nuevo objeto ni actualizar la referencia.

Tablas, matrices, vectores o arrays

En un array:

- Cada posición guarda un dato individual.
- Cada posición tiene un número asociado (un índice) que indica el lugar que ocupa dentro del array.
- Para acceder directamente a un dato individual (para darle valor, modificarlo o consultarlo...) lo hacemos mediante el nombre del array seguido del índice.

Un array lineal también se llama a veces **vector**, ya que sería conceptualmente parecido al

concepto matemático de vector. Los arrays bidimensionales son llamados también **matrices o tablas**. Podemos pensar en un array de cualquier dimensión, sin más que imaginarlo como un array de arrays. De hecho, así es como internamente construye Java los arrays de más de una dimensión.

```
int[ ] edadTrabajador;
edadTrabajador  =  new int [ 100 ];
```

Los ciclos for son los más adecuados para recorrer arrays, ya que sabemos exactamente las veces que hay que ejecutarlo, al conocer el valor inicial y el valor final del índice.

A todos los objetos de tipo array al ser creados se les asocia automáticamente una constante llamada **length** que indica el número de elementos que tiene ese array. El tamaño de un array puede decidirse en tiempo de ejecución, incluso ser aleatorio.

Para inicializar de forma explícita un array se le indica los valores que contiene entre llaves y separados por comas. Si es un array de más de una dimensión, cada nueva dimensión se expresa con otras llaves que contienen los elementos de esa nueva dimensión separados por comas.

```
String[][] matrizEscalera= { {"Miguel Ángel"},
                             {"Mª Carmen","Gonzalo"},
                             {"Francisco","Manuel Alberto","Manuel"}
                             };
```

Los valores de inicialización por defecto de los datos de un array serán:

- Para datos numéricos a 0 (0.0 si son reales)
- Para datos boolean a false
- Para datos por referencia (objetos) a null

UNIDAD 12: Estructuras estáticas de datos: Ficheros y manejo de Excepciones

Manejo de excepciones

En general existen dos planteamientos posibles, a la hora de manejar los errores:

- Prevenirlos: Supone incluir líneas de código adicionales en los puntos en los que pueden ocurrir los errores. La ventaja es que el programador que lee el código puede ver claramente si en ese punto se procesó o no el error, y si se comprobó o no correctamente. El inconveniente es que el código se "complica" con el procesamiento de errores y para el programador que lee la aplicación intentando comprender su funcionamiento, las líneas de control de errores le distraen de la lógica principal del sistema. Por ejemplo hemos usado la función length o el método length() para impedir accesos más allá del último elemento del array o el String
- Tratarlos adecuadamente una vez que ocurren: Una vez que el error (o excepción) ha ocurrido, se "captura", y se pasa el control del flujo del programa a una zona concreta que llamamos manejador de la excepción, de forma que se corrige ese error. La ventaja es que permite al programador quitar el código que maneja los errores del código principal, quedando un hilo de ejecución más limpio y claro, de forma que los programas así escritos son más fáciles de modificar y mantener. Además, es congruente con los criterios de modularidad, la tarea de tratar los errores es algo distinto al propósito principal del programa, y tiene sentido que se haga en un módulo distinto. El inconveniente es que las líneas de código que gestionan los errores pueden estar físicamente bastante separadas de las líneas en las que se produjo el error

¿En qué consiste manejar el error?

- Si el error no es excesivamente grave, se podrá dar otra oportunidad al usuario para corregirlo, por ejemplo introduciendo un nuevo valor para el número que se solicita, o volviendo a llamar al método que provocó el error, pero con otros parámetros más adecuados.
- Si el error es un fallo realmente grave, un error irrecuperable, al menos tendremos la oportunidad de avisar convenientemente al usuario de lo que ha pasado, podremos salvar lo que sea aprovechable de nuestra aplicación, y terminar el programa ordenadamente

Por **excepción** entenderemos una situación anómala en la ejecución de un programa, que impide que se siga ejecutando el flujo normal del programa, sin que en el ámbito en que se produce esa situación de error tengamos información suficiente para corregir el error, por lo que debemos pasar el control del flujo del programa a otro ámbito en el que quizás sea posible manejar ese error disponiendo de más información.

Así se propaga la excepción al método desde el que se invocó el código que generó la excepción, que si tampoco la sabe tratar la pasará a su vez al método que lo invocó, y así sucesivamente, hasta en el peor de los casos llegar al método main(), que le pasará la excepción a la máquina virtual Java, que escribirá un mensaje de error indicando el error ocurrido, escribirá la lista de invocaciones a métodos que han producido la excepción, y terminará la ejecución del programa.

En Java existe la clase **Throwable**, y cualquier situación anómala en la ejecución de un programa genera directa o indirectamente un objeto de la clase Throwable. La clase Throwable tiene dos subclases definidas en Java:

- **Error**. Indica que se ha producido un fallo irrecuperable, para el que es imposible recuperar y continuar la ejecución del programa. La máquina Virtual Java presenta un mensaje en el dispositivo de salida, y concluye la ejecución del programa. El programador no puede hacer nada

- **Exception**. Indica una situación anormal, pero que puede corregirse y reconducirse para que el programa no tenga que terminar forzosamente. Java además nos proporciona cerca de 70 subclases directas de la clase Exception, ya predefinidas, cada una para un tipo de error concreto.

Además de la inmensa cantidad de subclases de Exception que ya nos da definidas el lenguaje Java el programador puede definir, lanzar y usar sus propias excepciones en Java. En general, la sintaxis del manejo de excepciones en Java incluye el uso de las siguientes palabras reservadas: try, catch, finally, throw, throws

Capturar una excepción (try - catch – finally)

La parte de código que puede generar excepciones es el bloque de código que comienza con la palabra reservada **try**, seguida de un bloque de sentencias entre llaves, que son el código que puede generar el error.

El código al que se transfiere el control en el caso de que se produzca la excepción es el bloque iniciado por la palabra reservada **catch** seguida de un paréntesis que contiene la declaración del objeto de tipo Exception (o subclase de Exception) que se creará si se produce el error. Puede haber varios bloques catch para cada bloque try, cada uno encargado de una excepción

No es posible tener un bloque try sin ningún bloque catch o viciversa. Sólo se permite un bloque try sin ningún bloque catch si se ha incluido un bloque finally. Podemos colocar opcionalmente un bloque **finally**, que encerrará entre llaves un grupo de sentencias que se ejecutarán siempre, tanto si se produce una excepción como si no

Hay varias cosas que debemos tener en cuenta:

- Cuando hay una excepción que se lanza desde el bloque try, se transfiere el control al primer bloque catch cuyo parámetro coincida con el tipo de excepción que se ha lanzado.
- Si se coloca primero un bloque catch para una excepción más genérica, siempre se ejecutará ese bloque catch, y los que aparezcan detrás, correspondientes a subclases de excepciones de la anterior, no se alcanzarán ni se ejecutarán nunca
- En el manejador de la excepción nunca se produce un salto hacia atrás, a la sentencia que produjo la excepción.
- Si la excepción que se lanza desde el bloque try no se captura por ningún catch adecuado, se relanza hacia el método anterior en la cadena de llamadas, hasta llegar a un lugar donde se capture y se trate adecuadamente.
- En el peor de los casos será la propia máquina virtual la que se encargue de tratar la excepción,
- Si se escribe el bloque opcional finally, irá después del último catch, y se ejecutará siempre
- Si en las sentencias del bloque catch se produjera una nueva excepción, sería esta última la única que se propagaría hacia los métodos llamantes.
- Aunque se puede transferir el control desde dentro del bloque try hacia otras zonas del código, usando break, continue o return, no es demasiado aconsejable hacerlo

- En el bloque catch se recibe como parámetro el objeto Exception del tipo de la excepción que se ha producido en el bloque try, y se le pueden enviar una serie de métodos para obtener información sobre la excepción que se ha producido. Algunos de esos métodos son:
    - getMessage() , que recoge el mensaje de error asociado al tipo de excepción que se ha producido.
    - printStackTrace(), que escribe la cadena de llamadas que han generado la excepción, es decir el método que produjo la excepción, indicando desde qué otro método fue llamado, y a su vez desde qué otro método fue llamado éste, y así sucesivamente hasta llegar al método main(), desde el que se comienza la ejecución de todo el programa.

Para crear una **excepción propia** tenemos:

- Debemos definir una clase que declare la excepción que queremos crear como una subclase de alguna de las excepciones definidas por el lenguaje. Naturalmente puede ser una subclase de Exception.
- throw Se usa para indicar en qué lugar del código de nuestro método se lanza esa excepción, y para lanzarla, de hecho.
- throws Se usa en la cabecera del método para avisar a los programadores usuarios de nuestro método de que lanza un determinado tipo de excepción, y que deberán preocuparse de capturarla y manejarla convenientemente en los programas que usen ese método.

Ficheros

Un fichero o archivo es un conjunto de información sobre un mismo tema, tratada como unidad de almacenamiento y organizada de forma estructurada para la búsqueda y recuperación de un dato individual. Nuestra aplicación hará la petición de trabajar con ficheros tanto para lectura como para escritura, al Sistema Operativo, y éste se encargará de enmascarar los detalles de funcionamiento del dispositivo de entrada/salida,

La forma que implementa Java el trabajo con ficheros es mediante una abstracción más amplia, que es el concepto de **Flujo o Stream**. Los datos son algo que fluye en una corriente desde un origen (productor) hasta un destino (consumidor). Se considera dos tipos de flujos de datos:

- De entrada (InputStream). En ellos nuestra aplicación recibe los datos. Es el consumidor que retira los datos del flujo, que puede provenir de un teclado, o de un fichero, por ejemplo.
- De salida (OutputStream). En ellos nuestra aplicación produce los datos, y los envía al flujo para que lleguen hasta su destino, que puede ser el monitor, la impresora, o un fichero, por ejemplo.

En Java, en los flujos se escriben o leen bytes, en principio, de forma que cualquier información que quiero escribir en un flujo tengo que ir descomponiéndola en bytes, y escribiendo cada uno de ellos en el flujo de salida. De la misma forma, para recuperar esa información, tendré que ir leyendo uno a uno los bytes que me va proporcionando el flujo, y a partir de ellos recomponer la información original. El flujo es como una especie de "manguera unidireccional" que va metiendo la información byte a byte. La aplicación siempre ve el mismo extremo de la manguera, que siempre es igual. Pero la manguera en sí puede conectarse por el otro extremo a muy distintos orígenes o destinos de datos

Tanto **InputStream** como **OutputStream** son clases abstractas que definen una serie de

características comunes para todo flujo de entrada o de salida, respectivamente. El concepto de flujo permite independizar aún más la aplicación de los dispositivos de entrada o salida. Para que las operaciones de entrada/salida de nuestro programa funcionen usando otro dispositivo de entrada/salida o en otro sistema operativo (en cualquier otro sistema operativo) no tenemos que cambiar absolutamente nada en nuestra aplicación

Debes saber también que cuando ejecutas cualquier aplicación Java se crean automáticamente tres objetos que son flujos:

- **System.err** Es un objeto de tipo PrintStream, que a su vez es una subclase de FilterOutputStream, que a su vez es una subclase de OutputStream. En definitiva, es un flujo de salida definido en la clase System y que representa la salida de error estándar. Por defecto también es el monitor, aunque puede redireccionarse a otro dispositivo.
- **System.out** También es un objeto de tipo PrintStream. Es el flujo de salida definido en la clase System que representa la salida estándar. Por defecto también es el monitor, aunque puede redireccionarse a otro dispositivo.
- **System.in** Es un objeto de tipo InputStream, y está definido en la clase System como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.

Jerarquía de clases InputStream

InputStream

FileInputStream

- PipedInputStream
- ByteArrayInputStream
- StringBufferInputStream
- SequenceInputStream
- FilterInputStream
  - DataInputStream
  - LineNumberInputStream

BufferedInputStream

- PushbackInputStream

ObjectInputStream

Jerarquía de clases OutputStream

OutputStream

FileOutputStream

- PipedOutputStream
- ByteArrayOutputStream
- FilterOutputStream
  - DataOutputStream
  - PrintStream

BufferedOutputStream

- PushbackOutputStream

ObjectOutputStream

Jerarquía de clases Reader

Jerarquía de clases Writer

Reader

- CharArrayReader
- PipedReader
- StringReader

BufferedReader

- LineNumberReader
- InputStreamReader

FileReader

- FilterReader
  - PushbackReader

Writer

- CharArrayWriter
- PipedWriter
- StringWriter

BufferedWriter

- OutputStreamWriter

FileWriter

- FilterWriter
- PrintWriter

Las clases que nos permiten trabajar con ficheros en Java se encuentran disponibles en el paquete java.io, por lo que lo primero que debemos importarla así:

- import java.io.*;

Entre ellas se encuentra la clase **File**, que define una ruta hasta un fichero o un directorio, y nos permite consultar mediante toda una serie de métodos que define, información sobre el fichero o directorio que especifica esa ruta

El interface **FilenameFilter** se puede usar para crear filtros que establezcan condiciones de filtrado relativas al nombre de los ficheros. Establece en el método **accept()** la condición que debe cumplir un fichero de ese directorio para aparecer en el listado.

Existe también un interface **FileFilter**, que se usa para establecer filtros más generales, que afecten a otras características de los ficheros, además del nombre. La filosofía es la misma, y nos obliga a implementar también un método **accept()**.

La clase **FileInputStream** es la clase básica **para leer datos desde un fichero**. Sirve para leer todo tipo de datos de todo tipo de ficheros. Esta clase trabaja con bytes, es decir, se leen las informaciones por bytes desde el flujo, que estará asociado a un fichero.

Cuando una aplicación debe leer datos de un fichero, tiene que estar esperando a que el disco en el que está el fichero le suministre la información. Cualquier dispositivo de memoria masiva, por rápido que sea, es infinitamente más lento que la CPU del ordenador, resulta útil minimizar el número de accesos al fichero a fin de mejorar la eficiencia de la aplicación, y para ello se asocia al fichero una memoria intermedia (hace de intermediaria entre el fichero y la aplicación), de forma que cuando se necesita leer un byte del archivo, en realidad se traen hasta el buffer asociado al flujo (asociado a su vez al fichero) tanta información como le quepa. Mientras quede información en el buffer, nuestra aplicación leerá los datos directamente de la memoria principal, donde se encuentra el buffer, que es mucho más rápida que cualquier dispositivo de memoria masiva.

La clase básica que nos permite usar un fichero para salida o escritura de bytes en él, es la clase FileOutputStream. La filosofía es la misma que para la lectura de datos, pero ahora el flujo es en dirección contraria

Las subclases de **Writer y Reader** que permiten trabajar con ficheros de texto son:

- **FileReader**, para lectura o entrada desde un fichero de texto. Crea un flujo de entrada que trabaja con caracteres en vez de con bytes.
- **FileWriter**, para escritura o salida hacia un fichero de texto. Crea un flujo de salida que trabaja con caracteres en vez de con bytes.

Estas clases sólo se podrán usar para manejar ficheros de texto, y además es recomendable usarlas en este caso, porque son más eficientes. También se puede montar un buffer sobre cualquiera de los flujos que definen estas clases:

- **BufferedWriter** se usa para montar un buffer sobre un flujo de salida de tipo FileWriter.
- **BufferedReader** se usa para montar un buffer sobre un flujo de entrada de tipo FileReader.

Java proporciona un mecanismo para poder escribir y leer directamente objetos en los flujos asociados a los ficheros. Ese mecanismo recibe el nombre de **Serialización**.

- La Serialización consiste en la descomposición automática por parte de la máquina virtual de un objeto en una secuencia (serie) de bytes para poder escribirlos en un flujo asociado a un fichero.
- También es posible hacer el proceso contrario de "deserialización", que lee del flujo asociado al fichero la cadena de bytes, y a partir de ellos reconstruye el objeto original en memoria.
- Para que los objetos de una clase puedan ser serializados, **esa clase debe implementar el interface Serializable**.
- El interface **Serializable** no requiere implementar ningún método. Es sólo una "etiqueta" que debe colocársele a nuestra clase
- Al serializar un objeto, también se serializan todos los objetos que lo compongan, siempre y cuando también pertenezcan a clases serializables.
- Para serializar debemos usar flujos **ObjectOutputStream** y **ObjectInputStream**, sobre los que podremos escribir y leer objetos directamente.
- Como la serialización realmente transforma los objetos en cadenas de bytes, que son lo que se envía al fichero, **los flujos ObjectOutputStream deben montarse sobre un flujo FileOutputStream y los flujos ObjectInputStream deben montarse sobre un flujo FileInputStream**. No es posible usar Serialización con flujos de las jerarquías Writer y Reader.
- Los flujos ObjectOutputStream y ObjectInputStream proporcionan métodos **writeObject()** y **readObject()** para la ecritura y lectura de objetos de los flujos, pero además proporcionan toda una serie de métodos para trabajar con datos de tipo primitivo. Así tendremos **writeInt()** y **readInt()** permitirán escribir y leer valores int de los flujos.
- Al "deserializar" (leer los objetos del flujo) debemos aplicar un casting explícito a la clase del objeto que queremos leer. Esta clase debe estar accesible, y si no lo está se producirá una ClassNotFoundException, que deberá ser capturada convenientemente mediante un bloque try-catch.
- Sólo son serializables los objetos. Si necesitamos guardar en el flujo como parte importante de la información alguna variable de tipo static, debe guardarse de forma independiente, aunque en el mismo flujo.

UNIDAD 13: Estructuras dinámicas de datos: Punteros, listas, colas, pilas y árboles

Concepto de estructura de datos dinámica

Todas las **estructuras de datos estáticas** ocupan siempre la misma cantidad de memoria desde que se crean hasta que se destruyen. En general, podemos decir que las estructuras estáticas de datos no hacen un uso eficiente de la memoria.

En las **estructuras de datos dinámicas**, en cada momento se usa exactamente la memoria que se necesita para almacenar los datos que contiene. Las estructuras de datos dinámicas sí hacen un uso eficiente de la memoria, aunque en la mayoría de los casos es a costa de complicar los algoritmos de manipulación de estas estructuras (insertar, modificar, eliminar, buscar, procesar o listar elementos)

Punteros (Referencias en Java)

Por puntero entendemos un tipo de dato que corresponde a una dirección de memoria que a su vez referencia a un dato de otro tipo. Una variable de tipo puntero lo único que va a contener por tanto es una dirección de memoria (una referencia) que será la que realmente contendrá el dato.

Al construir una estructura de datos usando punteros, podemos hacer que su tamaño cambie y se ajuste en cada momento de forma exacta al número de elementos que contiene.

Un puntero apuntará a un nuevo elemento de la estructura, creado en cualquier lugar de memoria que estuviera libre, de forma que el tamaño de la estructura crece y la memoria ocupada por la estructura se ajusta al nuevo tamaño.

Al eliminar un elemento de la estructura, liberamos la memoria que ocupaba ese elemento y hacemos que el puntero que lo conectaba a la estructura apunte a nulo, es decir, que almacene un valor que indique que no apunta a ninguna posición de memoria, a ningún objeto. De esta manera, el tamaño de la estructura decrece, y la memoria que ya no usa queda libre.

En Java no existen los punteros, sino las **referencias**.

Una de las causas por las que resulta más fácil trabajar con referencias que con punteros es la existencia del recolector automático de basura ("garbage collector"). Cada vez que un objeto deja de tener alguna referencia que contenga su dirección, que lo apunte, no tenemos que hacer nada para liberar la memoria que ocupa.

Listas enlazadas

Java permite construir listas enlazadas de cualquier tipo de elemento u objeto, a condición de que el tipo de ese elemento haya sido definido adecuadamente en una clase, o por el propio lenguaje. Una lista es una colección de elementos colocados secuencialmente uno detrás de otro.

Las diferencias con los arrays son:

- En la lista enlazada cada elemento debe saber dónde está guardado en memoria el siguiente elemento de la lista.
- El último elemento de la lista debe saber indicar de alguna forma que detrás de él no hay ningún elemento más.
- Debemos tener una forma de llegar hasta el primer elemento de la lista, para a partir de él, poder recorrerla pasando al siguiente elemento, hasta llegar al último de la lista.

En la lista enlazada podemos tener cualquier número de elementos, desde cero hasta los que quepan en la memoria, y su tamaño crece y disminuye en tiempo de ejecución cada vez que se inserta o elimina un elemento de la lista, según las necesidades. La lista ocupa en cada momento exactamente el espacio que necesita.

Las operaciones básicas de las listas enlazadas son:

- ALTA: Inserción de nodos nuevos en la lista
- BAJA: Eliminación de un nodo de la lista.
- REINICIALIZAR: Eliminación de la lista completa.
- BÚSQUEDA: Búsqueda o consulta de los datos de un nodo concreto de la lista.
- PROCESAMIENTO: Recorrido y/o procesamiento completo de la lista.
- LISTADO COMPLETO: Listado o consulta de todos los nodos de la lista.
- LISTADO PARCIAL: Listado de los nodos que cumplan una condición.
- CONSULTA Nº NODOS: Consultar el total de elementos que contiene la lista.
- MODIFICAR: Modificar los datos de un nodo de la lista.
- GUARDAR: Guardar todos los datos de la lista en un fichero.
- CARGAR: Cargar desde un fichero los datos de la lista.

Otros tipos de listas enlazadas:

- **Listas doblemente enlazadas**
  - Cada nodo dispone de tres partes bien diferenciadas: una zona de datos, una referencia al siguiente elemento de la lista, y una referencia al elemento anterior en la lista
  - Habrá una referencia de tipo Nodo que apunte al primer nodo de la lista
  - Habrá una referencia de tipo Nodo que apunte al último nodo de la lista
  - Si la lista está vacía, tanto primerNodo como ultimoNodo apuntarán a null
  - El último nodo de la lista tendrá a null su campo siguienteNodo.
  - El primer nodo de la lista tendrá a null su campo nodoAnterior.
- **Lista enlazada circular**.
  - Una lista enlazada circular es realmente una lista enlazada "simple" en la que nos encargamos de que la referencia al nodo siguiente del último nodo de la lista apunte al primer nodo de la lista.
  - No obstante seguimos necesitando una referencia a un nodo de la lista para poder acceder a ella por algún punto. Podríamos llamar a esa referencia nodoEntradaActual.
- **Listas circulares doblemente enlazadas**
  - Básicamente es una lista doblemente enlazada en la que tenemos cuidado de que la referencia siguienteNodo del último nodo de la lista apunte siempre al primer nodo de la lista y de que la referencia nodoAnterior del primer nodo de la lista apunte siempre al último nodo de la lista
  - Eso significa que realmente no hay un primer nodo ni un último nodo, aunque seguimos necesitando una referencia a algún nodo de la lista que nos permita acceder a sus nodos. Esa referencia podría llamarse nodoEntradaActual.
  - Si la lista está vacía nodoEntradaActual debe apuntar a null.

Pilas

Es una estructura de datos secuencial, en la que los datos están almacenados en estricto orden de llegada de forma que el primero es el último introducido, en la que los elementos se introducen siempre en el mismo extremo de la lista, y se extraen siempre del mismo extremo de la lista. A ese extremo se le llama cabecera o cima de la pila. Es una estructura de datos tipo LIFO (Last In- First Out, o en buen romance, el último que entra es el primero que sale)

La implementación de pilas puede hacerse con arrays o con listas enlazadas. Las operaciones que debe tener disponibles el tipo Pila son las siguientes:

- **Meter (push): I**nsertar un nuevo dato en la cabecera de la pila.
- **Quitar, o sacar (pop):** Sacar un dato de la cabecera de la pila.
- **Pila Vacía:** Comprobar si la pila está vacía.
- **Pila llena:** Comprobar si la pila está llena.
- **Limpiar pila**: Sacar todos los elementos y dejar la pila vacía.
- **Cima:** Obtener el elemento de la cima de la pila.
- **Tamaño pila:** Número máximo de elementos que puede contener.
- **Elementos:** Consultar el número de elementos que tiene la pila.

Colas

Es una estructura de datos secuencial, en la que los datos están almacenados en estricto orden de llegada, de forma que el último introducido es el último de la cola, en la que los elementos se introducen siempre en el mismo extremo de la lista (final de la cola), y se extraen siempre del extremo contrario de la lista (cabecera de la cola). Es una estructura de datos tipo FIFO (First In- First Out, o en buen romance, el primero que llega es el primero en salir)

La implementación de colas puede hacerse mediante arrays o listas enlazadas. Las operaciones básicas son:

- **Añadir:** Insertar un nuevo dato en la cabecera de la cola.
- **Eliminar:** Sacar un dato de la cabecera de la cola.
- **Cola Vacía:** Comprobar si la cola está vacía.
- **Cola Llena:** Comprobar si la cola está llena.
- **Limpiar cola:** Sacar todos los elementos y dejar la cola vacía.
- **Cabecera o Frente:** Obtener el elemento de la cabecera de la cola.
- **Cola o Final:** Obtener el elemento del final de la cola.
- **Tamaño cola:** Número máximo de elementos que puede contener.
- **Elementos:** Consultar el número de elementos que tiene la cola.

Arboles

Para representar de forma conveniente las relaciones que existen entre los datos que se quieren almacenar, resulta útil disponer de una estructura de datos que represente esa relación de jerarquía o dependencia. Esa estructura jerárquica de datos son los **árboles**.

Existen también situaciones en las que los datos no se relacionan entre sí de forma jerárquica, pero conviene almacenarlos en un árbol como si lo hicieran, con el propósito de mejorar la eficiencia de los algoritmos de búsqueda, inserción o eliminación. Tal es el caso de los árboles de búsqueda.

Un árbol puede definirse de la siguiente forma:

- Es un conjunto, eventualmente no vacío, de elementos del mismo tipo llamados nodos.
- De entre todos los nodos existe un nodo "distinguido" al que llamamos raíz del árbol, de forma que dispondremos de una referencia que permanentemente apunte al nodo raíz.
- Los restantes elementos del árbol forman una colección de cero o más subárboles disjuntos entre sí.(los subárboles no comparten nodos)
- A los sucesores inmediatos de un nodo N se les llama hijos de N, y N se dice que es su padre. A los hijos de un mismo nodo se les llama hermanos.
- Cada nodo contendrá una parte de datos, que incluirá toda la información que realmente se quiere almacenar en la estructura, y varias referencias a cada arista de la que puede colgar un subárbol (cada uno de los posible nodos hijos). Esas referencias se usan para construir y mantener la estructura.
- Los nodos que no tienen hijos se llaman hojas.
- Un camino desde la raíz a una hoja se llama rama.
- El final de una rama se marca poniendo a null su referencia.
- La profundidad o altura de un árbol es el número de nodos que contiene la rama más larga.
- Cada nodo del árbol tiene asignado un número de nivel de la siguiente forma:
    - El nodo raíz tiene número de nivel 0.
    - Un nodo N distinto del raíz, tiene nº de nivel una unidad superior al de su padre o antecesor.

Si cada nodo puede tener un número grande de hijos o subárboles hay dos posibilidades:

- Si el número de subárboles posibles, aunque alto, es más o menos fijo y conocido. En este caso cada nodo incluye:
    - El campo (o campos) de datos.
    - Un vector de referencias a los posibles subárboles.
- Si el número de subárboles posibles además de alto es desconocido, pudiendo variar en un rango muy amplio (desde no tener ningún hijo hasta tener cientos o incluso miles de ellos): usar un vector sería muy ineficiente.. En este caso es preferible que cada nodo incluya:
    - El campo (o campos) de datos.
    - Una referencia al primer hijo de ese nodo.
    - Una referencia al siguiente hermano de ese nodo.

Tipos de árboles más usuales

- **Árboles binarios:** se define de la misma manera que hemos definido un árbol general n-ario, pero tiene la particularidad de que cada nodo puede tener como máximo dos nodos hijos, (n=2) que además están ordenados, y que solemos llamar hijo izquierdo e hijo derecho (o nodo izquierdo y nodo derecho).

- **Árboles binarios de búsqueda:** cada nodo cumple que su campo de información es mayor que el de cualquier nodo de su subárbol izquierdo y menor que el de cualquier nodo de su subárbol derecho. Esto nos permite mejorar las **búsquedas**, ya que en el nodo raíz podemos desechar la búsqueda en la mitad del árbol, y en cada nuevo nodo podemos volver a desechar la búsqueda en otra mitad del subárbol, y así sucesivamente. Haremos tantas comparaciones como niveles tenga el árbol. Se dice que la **eficiencia del algoritmo es del orden de $\log_2 n$**
- **Árboles binarios enhebrados:** hasta ahora las referencias al nodo izquierdo y derecho de los nodos hoja apuntan a null, y aproximadamente la mitad de las referencias de un árbol binario de búsqueda apuntan a null, inevitablemente. Los árboles enhebrados aprovechan esas referencias a null de los nodos para que apunten convenientemente a uno de los

nodos antecesores, de forma que sea posible mejorar la eficiencia al recorrer el árbol de una determinada manera. A estas referencias "reutilizadas" se les llama hebras, y a los árboles binarios que las contienen árboles enhebrados.

Debe ser posible distinguir las hebras de las referencias normales, añadiendo al nodo algún campo adicional que indique si la referencia es una hebra o una referencia normal, o indicando las referencias como enteros positivos y las hebras como enteros negativos, por ejemplo.

- Un valor null en un enlace derecho de un nodo p se reemplaza por una hebra al nodo que se visitaría después de p en un recorrido inorden (sucesor inorden de p)
- Un valor null en un enlace izquierdo de un nodo p se reemplaza por una hebra al nodo que se visitaría antes de p en un recorrido inorden (predecesor inorden de p)
- Crearemos un nodo adicional de cabecera. Será el padre del primer nodo y el sucesor del último nodo.

Mejoraramos la eficiencia de los algoritmos de recorrido del árbol y complicamos los algoritmos de inserción y borrado,

- **Árboles binarios en montón (heap)**
  - Un árbol en montón es un árbol binario completo, es decir, todos sus niveles tienen el número máximo de nodos posibles, excepto el último, en el que se cumple que los nodos están situados lo más a la izquierda posible.
  - Cada nodo cumple que el dato que almacena es mayor o igual que el dato de cualquier nodo perteneciente a cualquiera de los subárboles.
  - Para insertar un elemento, siempre lo insertamos como la última hoja del árbol, y comparamos repetidamente con el padre, de forma que si no cumplen la condición de montón se intercambian y se sigue comparando con el nuevo padre, hasta que se cumpla la condición de montón
  - Para borrar un elemento, se borra siempre el nodo raíz, y se sustituye por la última hoja. Se compara repetidamente con los hijos, de forma que si es menor que el mayor de los hijos, se intercambia con este, y se vuelve a comparar con los nuevos hijos, hasta que se cumpla la condición de montón,
  - Se puede definir análogamente un montón para establecer un orden de menor a mayor, cambiando la condición mayor o igual por menor o igual

- **Árboles B y B+:** Su principal aplicación es la gestión de las claves de los ficheros de índices de una base de datos. Las propiedades que los hacen muy adecuados para este fin es que el algoritmo de búsqueda de un valor dentro del árbol es extremadamente rápido. A eso se une la propiedad de que todos los nodos hojas, que son los que realmente contienen los datos, están en el mismo nivel del árbol, por lo que el tiempo de búsqueda es siempre el mismo, sea cual sea el elemento que estemos buscando y la posición que éste ocupe en el árbol. Además, las hojas del árbol pueden apuntar a las páginas en las que se divide el fichero índice para ser almacenado en disco, por lo que son muy adecuados para manejar índices muy grandes, que no caben en memoria, reduciendo al máximo el número de accesos a disco. Las características básicas de un árbol B de orden n son:
  - Cada nodo tiene un máximo de 2n claves y un mínimo de n. (Cada clave es un dato a guardar en el árbol)
  - Cada nodo que no sea una hoja, tiene m+1 hijos, siendo m el número de claves que contiene en ese momento el nodo
  - Todas las hojas están en el mismo nivel, lo que hace que el tiempo de búsqueda para un valor sea constante, con independencia del valor buscado.

Un árbol B+ consiste en combinar un fichero de datos estructurado en una secuencia de bloques de registros con un árbol B para indexar dichos bloques.

Recorrido de árboles binarios: Recorrido preorden, inorden y postorden

Existen tres formas básicas de recorrer un árbol binario:

- **Recorrido preorden: El nodo raíz se visita antes que los hijos.** Primero se visita, por ejemplo para escribir su valor, el nodo raíz, luego se recorre en preorden el subárbol izquierdo y finalmente se recorre en preorden el subárbol derecho. La condición de parada es que el puntero que apunta al raíz valga null. (Si no hay árbol que recorrer, pues no se recorre.)
- **Recorrido inorden: El nodo raíz se visita entre los dos nodos hijos.** Primero se recorre el subárbol izquierdo en inorden, luego se visita el nodo raíz, y luego se recorre el subárbol derecho en inorden. La condición de parada es que el puntero que apunta al raíz valga null.
- **Recorrido postorden: El nodo raíz se visita después de los dos nodos hijos.** Primero se recorre el subárbol izquierdo en postorden, luego se recorre el subárbol derecho en postorden, y finalmente se visita el nodo raíz. La condición de parada sigue siendo la misma, que el puntero que apunta al raíz valga null.

Es evidente que estos tres recorridos de un árbol son inherentemente recursivos, por lo que la implementación más habitual de estos algoritmos es usando recursividad.

UNIDAD 14: Programación orientada a objetos en Java

Paradigma de programación orientada a objetos

Cuando se desarrolla un programa informático siempre se busca que se trate de un producto de calidad. Para que un software sea de calidad necesita:

- **Factores Externos:** pueden ser detectados por los usuarios.
    - Definimos **corrección** como la capacidad de los productos software para realizar con exactitud sus tareas, tal y como se definen en las especificaciones, y es una de las características que debe reunir un programa para que se considere un producto de calidad.
    - Definimos **robustez** como la capacidad de los sistemas software de reaccionar apropiadamente ante condiciones excepcionales. Un programa que sea capaz de detectar y reaccionar a este tipo de situaciones no previstas
    - Definimos **extensibilidad** como la capacidad de los sistemas software de ser adaptados fácilmente a los cambios de las especificaciones
    - Definimos **reutilización** como la capacidad de los elementos del software de servir para la construcción de muchas aplicaciones diferentes
    - Definimos **compatibilidad** como la capacidad de combinar unos elementos software con otros,
    - Definimos **eficiencia** como la capacidad de un sistema software de requerir la menor cantidad posible de recursos hardware
    - Definimos **portabilidad** como la capacidad que tienen algunos productos software para funcionar en diferentes entornos hardware y software,
    - Definimos **facilidad de uso** como la capacidad de los programas para que personas con diferentes formaciones y aptitudes puedan aprender a usarlos y explotarlos
    - Por último, otra característica que siempre vamos mirando los usuarios, si no es la primera, es el precio de los productos, aunque sea éste un factor que no tiene por qué tener relación alguna con la calidad del producto.
- **Factores Internos:** perceptibles sólo por profesionales de la informática que tienen acceso al código fuente de los programas.
    - Seguir una **metodología modular** en el desarrollo de programas informáticos ayuda a producir sistemas software a partir de elementos autónomos interconectados por una estructura simple y coherente, y esto favorece la consecución de los dos factores de calidad externos del software principales: la extensibilidad y la reutilización.
    - Es importante que los programas sean **legibles**. Ssto comprende que sus líneas de código tengan buenos comentarios que permitan entender rápidamente qué hace el programa y que estén correctamente tabuladas para su rápida comprensión.

Un paradigma de programación representa un enfoque particular o filosofía para la construcción de software. Cada paradigma ofrece un estilo de acercarse a los problemas y deriva en una manera particular de abordarlos y resolverlos. Paradigmas de programación hay muchos, si bien son dos los más importantes:

- El paradigma **imperativo**, que da lugar a la programación estructurada o basada en procedimientos.
- El paradigma **orientado a objetos**, que da lugar a la programación orientada a objetos.

Las principales dificultades y problemas surgidos en las técnicas estructuradas parten de su propio enfoque:

- Las metodologías estructuradas hacen una **división de procesos y datos**, donde los procesos son los que guían la lógica del programa.
- Además, **las dependencias existentes entre datos y procesos quedan reflejadas en el programa**, lo que implica que cualquier cambio en el proceso o en los datos suponga cambios importantes en el propio programa. Esto hace que los **programas** sean **poco extensibles**.
- Pero el problema principal de la programación estructurada o basada en procedimientos es que **las unidades de programación no reflejan de manera fácil y efectiva a las entidades del mundo real**, lo cual deriva en que estas **unidades no** sean particularmente **reutilizables**. Con gran frecuencia los programadores deben comenzar "de nuevo" cada nuevo proyecto y escribir código similar "desde cero".

Si percibimos el mundo real como un mundo formado por objetos, entonces considerar los problemas en términos de objetos nos resultará más sencillo. Además, mediante la tecnología de objetos, las entidades de software creadas si se diseñan apropiadamente, tienden a ser mucho más reutilizables en proyectos futuros. La programación orientada a objetos tiende a producir software que es más comprensible, mejor organizado y fácil de mantener, modificar y corregir.

Lo que define a los objetos es:

- Todos los objetos tienen atributos que los describen y todos los objetos de la misma clase o tipo vendrán definidos por los mismos atributos
- Todos los objetos exhiben un comportamiento o realizan operaciones que especifican lo que hacen o para qué sirven y todos los objetos de la misma clase o tipo tendrán el mismo comportamiento.

Toda la potencia del paradigma de programación proviene precisamente de esta correspondencia directa entre el dominio del problema y el dominio de la solución; es decir, la correspondencia directa entre la realidad modelada y el programa que la modela. La programación orientada a objetos nos brinda una forma natural de ver el proceso de programación, a saber, mediante el modelado de objetos reales, sus atributos y su comportamiento.

Con la teoría orientada a objetos, construimos la mayoría del software del futuro mediante la combinación de "partes estándares e intercambiables" llamadas clases. Para construir los objetos haremos uso de las técnicas de la programación estructurada

Clases

Una clase es una plantilla que define la forma de un tipo de objetos. En esta plantilla se especifican los atributos y el comportamiento con los que van a contar los objetos que se construyan a partir de dicha plantilla. El hecho de crear un objeto a partir de una clase recibe el nombre de instanciar un objeto. Los objetos son las instancias de las clases, cumpliéndose durante la ejecución de una aplicación que:

- Todo objeto es instancia de una única clase.
- Toda clase que forma parte del programa tiene, en un instante dado, cero o más objetos que son instancia de ella.

Un programa orientado a objetos es:

- Una colección estructurada de clases que definen los distintos tipos de objetos que van a intervenir en la resolución del problema.
- Una especificación de qué objetos concretos, cuántos y de qué tipo se van a utilizar en la resolución de un problema y cómo van a colaborar dichos objetos para ello.

Los objetos no existen hasta que el programa no empieza a ejecutarse. A partir de ese momento los objetos empiezan a crearse, a interactuar y a desaparecer según indique el propio programa. Por su parte, cuando el programa está en ejecución lo único que existe son los objetos. Por eso se dice que las clases representan la parte estática de la aplicación.

Desde el punto de vista semántico, una clase es un mecanismo de definición de nuevos tipos de datos, donde al mismo tiempo que se describe una estructura de datos para representar valores de un dominio, también se describen las operaciones aplicables sobre las entidades de dicho tipo de datos. Así, los componentes o miembros de una clase son los siguientes:

- Los atributos. Son las características y la estructura de almacenamiento que tendrán los objetos de la clase.
- Los métodos. Son el conjunto de funcionalidades que describen la naturaleza y el comportamiento que tendrán los objetos de la clase; es decir, las operaciones aplicables a dichos objetos.

Una clase bien diseñada debe definir una única entidad, que agrupe todas sus características y comportamiento, pero que sea una única entidad del mundo o problema que se está modelando.

Las clases, aunque deben presentar una alta cohesión y un bajo acoplamiento, tienen ciertas relaciones entre ellas, de forma que un programa bien estructurado contará con clases bien diseñadas que se relacionen entre sí. Las relaciones entre clases se reducen a dos tipos:

- **Clientela** (A veces nombrada **Composición**): Una clase B es cliente de una clase A si B contiene una declaración en la que se establezca que cierta entidad de B, ya sea ésta un atributo, parámetro o variable local de alguno de sus métodos, es de tipo A.
- **Herencia**: Una clase B hereda de una clase A cuando incorpora la estructura (atributos) y el comportamiento (métodos) de la clase A, pero puede incluir algunas adaptaciones, como añadir nuevos atributos o nuevos métodos. Se dice entonces que la clase B es una versión especializada de la clase A, o más comúnmente, que la clase B es una subclase de la clase A o que deriva de la clase A y, por otra parte, se dice que la clase A es superclase de las clases B y C.

Definición de clases en java

A la hora de definir una clase en Java debemos tener en cuenta los siguientes aspectos:

- La definición e implementación de una clase en Java se realiza en el mismo archivo.
- El archivo de una clase Java debe tener el mismo nombre que la clase principal que contenga el archivo.
- En Java, la palabra clave class es la que nos va a permitir definir una clase.
- En la definición de la clase se deben incluir los atributos que contiene y los métodos que operan sobre ellos.

La definición de una clase en Java va a constar de las siguientes partes básicas:

- **Cabecera de la clase.** En la cabecera de la clase se indica el nombre de la clase y una serie de características de la misma, como son: la clase de la que deriva (es decir, su superclase), los privilegios de acceso a la clase y si la clase implementa, o no, uno o varios interfaces.Además, por convención, el nombre de la clase debe empezar con una letra mayúscula. Como hemos dicho anteriormente, en la cabecera se indica tanto el nombre de la clase, como una serie de características de la clase que vienen determinadas por lo que se conoce con el nombre de modificadores de la clase. Los 3 primeros modificadores se colocan delante de la palabra "class" y los 2 últimos detrás:
    - La palabra clave **public**: cualquier clase puede hacer uso de ella o relacionarse con ella. Cuando no aparece la palabra clave public, entonces se dice que la clase es no pública y su nivel de acceso es de tipo package (de tipo paquete), lo que quiere decir que sólo pueden relacionarse con ella o hacer uso de ella las clases pertenecientes a su mismo paquete. En un mismo archivo no puede haber definida más de una clase pública, aunque sí pueden coexistir otras clases no públicas.
    - La palabra clave **final**. Indica que con esta clase se termina la cadena de herencia; es decir, no va a haber clases que hereden de ella.
    - La palabra clave **abstract**: estas clases que pueden tener algún método que esté declarado pero para el que no se dé su implementación, la cual será aportada por las clases que hereden de ella. Los conceptos de clase final y clase abstracta son excluyentes y las palabras clave final y abstract no pueden aparecer juntas como modificadores de una clase.
    - La palabra clave **extends**. Indica que la clase hereda de otra clase, llamada superclase o clase madre
    - La palabra clave **implements**. Indica que la clase implementa los métodos de una o varias interfaces.

      [public] [final | abstract] class Nombre_de_la_clase [extends superclase]
      [implements Interface_1 [, Interface_2] [, Interface_3] ...] {

- **Cuerpo de la clase.** En el cuerpo de la clase se incluye el contenido de la clase; es decir, se definen sus atributos y se declaran e implementan sus métodos. A los atributos de una clase se le añaden una serie de modificadores que indican una serie de características del atributo.

      [private | protected | public] [static] [final] [transient] [volatile]
      tipo_atributo Nombre_atributo;

  La definición de cada método de una clase está formada por dos partes:
    - la cabecera del método, que es donde se declara. Formada por:
        - Modificadores del método que indican una serie de características del método
          [private | protected | public] [static] [abstract] [final] [native] [synchronized]
        - Tipo de datos del valor devuelto por el método.
        - Nombre del método por convención debe comenzar por minúscula.
        - Parámetros del método. Entre paréntesis y separados por comas. Los parámetros se pasan valor, aunque en el caso de objetos se pasan por referencia
        - Las excepciones. Deberá indicarse qué excepciones va a generar el método. Se usa la palabra clave throws, seguida de una lista con las excepciones
    - el cuerpo, que es donde reside su implementación.

Introducción a la modularidad, el encapsulamiento y la ocultación de la implementación

Como programadores, lo que estamos haciendo al definir una clase, no es más que encerrar o encapsular en una misma entidad conceptual o semántica, la clase, todos los datos que describen a una entidad genérica del mundo real, junto con todas las operaciones que se pueden realizar sobre esos datos. Pero para que el encapsulamiento sea completo, éste no tiene que ser sólo semántico, sino que debería quedar reflejado también a nivel sintáctico en el propio programa, en su código. Es decir, el lenguaje debe permitir, e incluso obligar, a que cada unidad semántica o clase esté aislada en un único módulo sintáctico del programa.

La propiedad de encerrar en una misma entidad sintáctica o módulo la totalidad de una entidad semántica, clase o tipo de dato, recibe el nombre de **encapsulamiento** y es una de las características fundamentales de la programación orientada a objetos. Además al estar cada clase en un módulo del programa y al contener cada módulo del programa una única clase, estamos consiguiendo la descomposición **modular o modularidad**

Conseguimos entonces soporte para poder hacer uso de la que probablemente sea la piedra filosofal de la programación orientada a objetos: **la ocultación de la implementación o programación orientada a la interfaz**.

Al cliente de una clase:

- Le interesa conocer qué información almacena la clase de la que es cliente y, por tanto, qué información puede proporcionarle ésta, pero no le interesa conocer en absoluto cómo dicha información está representada internamente en la clase
- Le interesa conocer qué servicios le ofrece la clase de la que es cliente, qué hacen los distintos métodos que proporciona, pero no le interesa conocer cómo están implementados dichos métodos, ni su código ni los algoritmos que utilizan los métodos para llevar a cabo sus tareas.

Es decir, en la programación orientada a objetos prima el "qué" sobre el "cómo", la interfaz sobre la implementación, donde la interfaz de una clase especificará cómo se utilizan los servicios o funcionalidad ofrecida por la clase, indicando para cada servicio:

- cómo se invoca o solicita,
- qué hace,
- qué información, si alguna, necesita de entrada y
- que información, si alguna, devuelve o proporciona como salida.

Por su parte, la implementación de una clase se oculta a sus clientes, que pueden utilizarla simplemente conociendo su interfaz. Esto es lo que se conoce en el mundo de la orientación a objetos como **ocultación de la implementación**, y es una filosofía de programación que proporciona, principalmente, las siguientes ventajas:

- Simplifica la percepción del cliente con respecto a una clase
- Hace mucho más sencillo el mantenimiento de los programas, al hacer mucho menos traumática la modificación de los mismos.

Existen unidades mayores que la clase para la encapsulación y la ocultación de la información y se llaman **paquetes**. Un paquete no es más que un conjunto de clases que tienen alguna relación entre sí y que, por ello, se decide que confíen unas en otras, donde esta relación de confianza implica que se va a permitir que puedan conocer unas las "interioridades" de las otras.

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

- Una clase no tiene por qué pertenecer a un paquete.
- De aparecer la sentencia de definición, ésta debe aparecer al principio del archivo
- Una clase puede pertenecer a un único paquete
- Pertenecerán a un paquete todas las clases que comiencen con una sentencia de declaración de paquete y cuyo nombre de paquete en dicha sentencia coincida.

Java también soporta el concepto de jerarquía de paquetes. Esto es parecido a la jerarquía de directorios de la mayoría de los sistemas de ficheros de un sistema operativo, donde: las clases serían el equivalente a los ficheros de un sistema de ficheros; un paquete sería una carpeta que contiene clases; un paquete, en vez de clases, puede contener otros paquetes. para localizar una clase en la jerarquía de paquetes daremos la ruta desde el paquete raíz de la jerarquía, separando cada uno de los paquetes de dicha ruta con un punto.

El mayor beneficio que se obtiene de esta manera de organización de las clases en paquetes es que proporciona una convención para tener nombres de clases que sean únicos.

Para no tener que anteponer toda la ruta para acceder a un paquete podemos importar el paquete o la clase haciendo uso de la sentencia import, de la siguiente manera:

```
import Nombre_de_Paquete;
```

Debemos tener en cuenta:

- La sentencia import debe aparecer al principio de la clase, justo después de la sentencia package, si ésta existiese.
- Si queremos importar de una vez todas las clases de un paquete ponemos *
- Deberá aparecer una sentencia import por cada clase o paquete importado.

Para expresar el nivel de ocultación de los métodos y atributos de las clases usamos modificadores. Ordenados de más restrictivo a menos restrictivo, son:

- La palabra clave **private**. Es el nivel de acceso más restringido e indica que al atributo o método que vaya precedido por este modificador sólo se podrá acceder desde la propia clase.
- La palabra clave **package**. Se usa por defecto e indica que el atributo o método podrá ser accedido desde cualquier clase del mismo paquete, pero no desde una clase perteneciente a otro paquete.
- La palabra clave **protected**. Indica que al atributo o método sólo se podrá acceder desde la propia clase, desde sus subclases y desde las clases que pertenezcan a su mismo paquete.
- La palabra clave **public**. Es el nivel de acceso menos restrictivo e indica que al atributo o método se podrá acceder desde cualquier otra clase.

Métodos públicos de acceso a atributos privados

- Deberíamos utilizar el modificador public para todos aquellos atributos o métodos de la clase que vayan a formar parte de la interfaz de la clase; es decir, para presentar a los clientes de la clase una vista de los servicios que la clase proporciona.
- Deberíamos utilizar el modificador private para el resto; es decir, para ocultar toda aquella información (atributos y métodos) de la clase que no pertenece a la interfaz de la misma

- Los atributos de una clase nunca deben pertenecer a la interfaz de la misma y, por lo tanto, deberán ser declarados como privados.
- Generalmente los métodos de la clase pertenecerán a la interfaz de la misma y, por lo tanto, deberán ser declarados como públicos.
- Es posible que tengamos la necesidad de tener métodos privados dentro de una clase. Estos métodos reciben el nombre de **métodos utilitarios o métodos ayudantes**, ya que pueden ser llamados sólo por otros métodos de esa clase y su misión es la de ayudarles a realizar sus tareas internas.

Las clases proporcionan a menudo métodos públicos para permitir a los clientes de la clase establecer (escribir) u obtener (leer) directamente los valores de los atributos privados:

- Los métodos "establecer", o métodos "set" en inglés, también se conocen comúnmente como métodos mutadores
- Los métodos "obtener", o métodos "get" en inglés, también se conocen comúnmente como métodos de acceso o métodos de consulta.
- Por convención dichos métodos se nombran con la palabra set o get, o sus correspondientes en castellano, junto al nombre del atributo que modifican o consultan.
- No es obligatorio que exista un método "set" y un método "get" para cada atributo privado de la clase, éstos sólo deberán proporcionarse cuando tengan sentido.
- Si una clase tiene algún atributo declarado como público, cualquier método de cualquier clase podrá leer o modificar su contenido. Esto supone una violación del principio de ocultación de la información y de la programación orientada a la interfaz
- Los métodos set y get servirán para ocultarles la verdadera implementación de esa información en la clase, que "verán" el atributo única y exclusivamente a través de estos métodos.
- Estos métodos servirán para controlar la manera en la que los clientes pueden tener acceso al atributo, impidiendo que algún agente externo pueda introducir en los atributos valores no válidos para los mismos.

El ocultamiento de la información fomenta la capacidad de modificar los programas y simplifica la percepción que tiene el cliente acerca de una clase.

Objetos o instancias de una clase

Acerca de los objetos podemos decir:

- Para obtener un objeto debe existir una clase a partir de la cual instanciarlo
- Instanciar un objeto no es más que crear un objeto **concreto** a partir de una clase
- Todo objeto es instancia de una única clase y toda clase que forma parte del programa tiene, en un instante dado de la ejecución del mismo, cero o más objetos que son instancia de ella.
- Un programa orientado a objetos es una colección estructurada de clases que definen los distintos tipos de objetos que van a intervenir en la resolución del problema
- Los objetos no existen hasta que el programa no empieza a ejecutarse.
- Un objeto es una instancia de una clase, creada en tiempo de ejecución

Una referencia es un valor en tiempo de ejecución que está o vacío o conectado. Si está conectado, una referencia identifica a un único objeto. Nada más crear una referencia, está se encuentra vacía; es decir, el valor inicial de la referencia al objeto es null. Una vez declarada la referencia al objeto es el momento de crear la instancia de la clase es el momento de hacer que la referencia pase al estado "conectada"

nombre_objeto = new constructor_de_la_clase(parámetros_del_constructor);

- El operador new se encarga de reservar memoria suficiente para crear una instancia al objeto, asignando a la referencia la dirección de memoria en la que se encuentra el objeto recién instanciado.
- El método constructor de la clase es un método especial de la misma que se ejecuta exclusivamente en el instante de la creación de una instancia u objeto y que se encarga de inicializar dicho objeto

Una vez creado un objeto, éste tiene identidad propia que lo distingue de los demás. Internamente, para distinguir entre los objetos, el lenguaje lo que hace es asignar a cada objeto un identificador único, llamado oid o identificador de objeto.

Cada objeto que se instancia tiene su propia zona de almacenamiento en memoria donde está almacenada una copia, de uso exclusivamente suya, de los atributos que caracterizan a un objeto de su clase. Los atributos o variables que son propios de cada objeto se les conoce con el nombre de **variables de instancia.** A los métodos que hacen uso de estas variables de instancia, ya sea para su consulta o modificación, reciben el nombre de **métodos de instancia.** Tanto a las variables como los métodos de instancia, reciben el nombre de **miembros de objeto.**

Una **variable de clase** representa información en toda la clase; es decir, es una variable que será compartida por todos los objetos de la clase. Para crear una variable de clase haremos uso del modificador **static** cuando declaremos la variable. Aunque las variables de clase pueden parecer variables globales, tienen alcance a nivel de clase. También existen **métodos de clase,** que junto a las primeras forman lo que se conoce como miembros de clase que se usan solamente para acceder a las variables de clase. Se puede acceder a los miembros de clase a través de una referencia a cualquier objeto de dicha clase, de igual manera que se accede a cualquier otro miembro de la clase. Además estos métodos pueden utilizarse incluso aunque no se haya instanciado objetos de esa clase. Expertos programadores recomiendan acceder siempre a los miembros de clase de esta última manera. Los miembros de clase, tanto las variables como los métodos, también pueden ir acompañados de un modificador de control de acceso, principalmente el modificador public o el modificador private

Debemos tener en cuenta:

- La asignación entre referencias no implica copia de valores sino de referencias. La posibilidad de tener referencias múltiples a un mismo objeto se llama **aliasing.** Si varias referencias apuntan a un mismo objeto, si cualquiera de ellas cambia un valor del objeto, cambiará para todas las referencias
- Si queremos duplicar un objeto; es decir, queremos obtener una copia de un objeto que sea independiente del objeto original, hay que definir un método duplicador
- Cuando se pasa un objeto como parámetro a un método, lo que se va a pasar al interior del método es la referencia al mismo, con la implicación que esto tiene: cualquier cambio que se haga en el parámetro va a hacerse en la referencia, con lo que el cambio quedará registrado en el objeto incluso al salir del método.
- Cuando comparamos dos referencias no estamos comparando los valores de los objetos a los que apuntan, sino las propias referencias
- Para poder realizar la comparación entre objetos por los valores de sus atributos, deberemos hacer lo mismo que para la copia: definir un método comparador

Métodos o Mensajes

Una vez instanciados los objetos de la clase, para interactuar con esos objetos y modificar su

estado, debemos provocar la ejecución de los métodos públicos que dichos objetos ofrecen:

- Provocar la ejecución de un método de un objeto se le llama "realizar una petición", "solicitar un servicio" o "enviar un mensaje" al objeto
- Desde el punto de vista del objeto, recibir la solicitud para que ejecute uno de sus métodos recibe el nombre de "recibir una petición", "recibir una solicitud de servicio" o "recibir un mensaje".
- El emisor de un mensaje será siempre un objeto e, igualmente, el receptor de un mensaje será otro objeto. Por lo tanto, un mensaje no es más que una forma de comunicación o de colaboración entre los objetos.

Un programa orientado a objetos no es más que una colección de objetos que intercambian mensajes unos con otros con el objetivo de llevar a buen puerto el cometido para el que el programa está hecho.

Java permite declarar en la misma clase varios métodos distintos con el mismo nombre, siempre y cuando éstos tengan distintos conjuntos de parámetros, a esta técnica se le llama **sobrecarga de métodos.** La sobrecarga se utiliza para crear varios métodos con el mismo nombre que realizan tareas similares, pero en tipos de datos distintos. Los métodos sobrecargados se distinguen por su firma: una combinación del nombre del método, el número y tipo de sus parámetros y no puede haber en una misma clase dos métodos que se llamen igual y que además tengan el mismo número y tipo para sus parámetros.

Constructores

A la acción de dar un valor inicial a las variables recién declaradas se le denomina **inicialización**. Es recomendable inicializar los objetos que se creen antes de usarlos, y es ahí donde intervienen unos elementos que reciben el nombre de métodos constructores o, simplemente, **constructores**.

El método constructor de la clase es un método especial de la misma que se ejecuta exclusivamente en el instante de la creación de una instancia u objeto y que se encarga de inicializar dicho objeto, es decir, de asignar los valores iniciales a todos sus atributos. Si no se inicializan los atributos el compilador lo hará poniendo: el valor 0 para los tipos numéricos primitivos, el valor false para los valores boolean, y el valor null para las referencias.

Debemos tener en cuenta:

- Se invoca única y exclusivamente inmediatamente después de la creación del objeto
- El constructor de una clase obligatoriamente debe tener el mismo nombre que la clase
- Un constructor no puede especificar un valor de retorno
- Aunque el lenguaje Java lo permite, es recomendable no dar a ningún método, que no sea constructor, el mismo nombre que la clase.
- Los constructores se declaran con un acceso público
- Un constructor puede definir parámetros en su declaración
- Estos argumentos o parámetros de los métodos constructores reciben el nombre de inicializadores, pues permitirán al programador indicar, en el momento de la creación del objeto, qué valores concretos debe establecerse para los atributos del objeto. También es posible para el programador proporcionar un constructor sin parámetros
- Se requiere que toda clase tenga cuando menos un constructor. Si no se declarasen constructores para una clase, y sólo en ese caso, el compilador crearía un constructor predeterminado que no tomaría argumentos.
- Un constructor puede llamarse sin argumentos sólamente si no hay constructores declarados para la clase, en cuyo caso se llama al constructor por defecto, o si hay un

constructor público sin argumentos.
- Aunque se pueda llamar a otros métodos de la clase desde un constructor, no es recomendable hacerlo.
- Los constructores pueden sobrecargarse premitiendo así a los objetos de una clase inicializarse de distintas formas.

Uso del operador this y del método this()

La **referencia implícita al objeto**, hace referencia al propio objeto pero sin mencionarlo expresamente. Sin embargo, aunque no sea obligatorio, es posible reflejar esta referencia al propio objeto de forma explícita mediante la palabra clave this. Todo objeto puede hacer referencia a sí mismo mediante la palabra clave this, la cual hace referencia al propio objeto en el que se está ejecutando la sentencia. Para un método en el cual un parámetro o variable local tenga el mismo nombre que una variable de instancia de la clase, debe usar obligatoriamente la referencia this si desea tener acceso a la variable de instancia; de no ser así, estará haciendo referencia al parámetro o variable local del método.

Serçia un error de sintaxis que un método static llame a un método de instancia de manera directa o acceda a una variable de instancia directamente, incluido a través de la referencia this, al no haber objeto de la clase al cual hacer referencia.

A parte. El método this() se usa para hacer referencia dentro de un constructor de una clase a otro constructor sobrecargado de la misma clase, aquél que coincida con la lista de parámetros de la llamada. De usarse el método this(), forzosamente tiene que ser la primera línea de código del constructor.

Introducción al concepto de Interface

La programación orientada a objetos es una programación orientada a la interfaz que pretende ocultar la implementación; es decir, es una filosofía de programación que se centra en el qué se hace en vez de en el cómo se hacen las cosas. La parte pública de la clase configura lo que llamábamos **interfaz** de la clase y es la parte con la que interactuará cualquier entidad que quiera utilizar algún objeto de dicha clase.

Llamémosle **interface**, la cual especifica el comportamiento que debe tener todo artefacto que quiera ser considerado "de un determinado modo"

UNIDAD 15: Herencia y polimorfismo.

Reutilización y extensibilidad: pilares básicos de la calidad del software

- La **extensibilidad** o capacidad de los sistemas software de ser adaptados fácilmente a los cambios de las especificaciones, produciendo programas a los que se les pueda añadir nueva funcionalidad o se les pueda realizar modificaciones de manera sencilla y poco traumática, lo cual reduce considerablemente los costes derivados del mantenimiento.
- La **reutilización** o capacidad de los elementos del software de servir para la construcción de muchas aplicaciones diferentes, evitando así la reescritura del mismo código una y otra vez.

Para que un software sea así hemos visto:

- Una clase modela una entidad del mundo real, por lo que si ésta está diseñada apropiadamente, es muy probable que pueda ser reutilizada sin ninguna modificación en proyectos futuros
- Una clase es un módulo tanto a nivel semántico (se encierra en una misma entidad conceptual) como a nivel sintáctico (está encerrada en un único archivo o módulo a nivel de programa)
- La propiedad de la programación orientada a objetos conocida con el nombre de ocultación de la implementación, favorece la extensibilidad, pues permite variar la implementación de las clases sin afectar a sus clientes

Pero se necesita más para alcanzar en su totalidad los objetivos de reutilización y extensibilidad:

- Se necesitan técnicas para capturar las evidentes semejanzas que existen dentro de grupos de estructuras similares, para no tener que estar repitiendo lo mismo una y otra vez, pero respetando al mismo tiempo las muchas diferencias que caracterizan a cada una de ellas.
- Se necesitan técnicas que me permitan programar para esta generalidad común, de tal manera que el mismo código funcione exactamente igual con cada uno de los distintos elementos particulares que comparten dicha generalidad

Herencia

Las relaciones entre clases se reducen a dos tipos:

- La composición, que no es más que un caso particular del concepto de clientela. Se da cuando una clase tiene alguna variable de instancia que es una referencia a un objeto de otra clase.
- La herencia, que es quizá una de las características principales de la programación orientada a objetos. Una de ellas no es más que una especialización o una extensión de otra.

La herencia es un mecanismo potente de reutilización, pues una clase absorbe las características de otra ya existente, la cual reutiliza. Algunas cuestiones importantes:

- Una superclase puede tener muchas subclases,
- Cuando una clase hereda de otra, hereda todos y cada uno de los miembros de la misma
- Cada subclase puede convertirse en superclase de futuras subclases. Llamamos

superclase directa a la superclase a partir de la cual la subclase hereda en forma explícita. Por su parte, llamamos **superclase indirecta** a una clase de la que no se hereda directamente, sino a través de otra clase que sí hereda de ella directa o indirectamente. UIna subclase posee todas las características de todas las clases de las que hereda

- Una clase puede heredar directamente de varias clases a la vez. e llama **herencia múltiple** al hecho de que una clase herede de forma directa de dos o más clases.
- Al ordenamiento que define y refleja las relaciones de herencia entre clases recibe el nombre de **jerarquía de clases o jerarquía de herencia**
- Un objeto de la subclase es también un objeto de la superclase y, consecuentemente, puede ser tratado como tal
- Un objeto de la superclase no puede ser tratado como un objeto de la subclase
- Un problema que se puede producir en la herencia es que una subclase herede métodos o propiedades que no necesita o que no debe tener, lo cual tiene difícil solución.
- Otro problema que puede aparecer con la herencia es que una subclase requiera que un método heredado realice su tarea de una manera específica o distinta a como lo hace en la superclase. En estos casos la subclase puede sobrescribir el método de la superclase con una implementación propia y apropiada. Esta acción recibe en el ámbito de la programación orientada a objetos el nombre técnico de **redefinición de un método**

Para identificar las jerarquías los programadores expertos suelen utilizar dos estrategias:

- Tratar de detectar clases que, aunque sean distintas, compartan un comportamiento común. Esta estrategia recibe el nombre de **generalización o factorización**.
- Tratar de detectar clases que sean un caso especial de otras. Esta estrategia recibe el nombre de **especialización o abstracción**.

Para establecer en Java que una clase hereda de otra clase ya existente, en la definición de la subclase utilizaremos el modificador extends en la cabecera de su definición, seguido del nombre de la clase que va a ser su superclase, de la siguiente manera:

```
public class TrabajadorFijo extends Trabajador{
}
public class TrabajadorTemporal extends Trabajador{
}
```

Java no soporta herencia múltiple. Todas las clases en Java, excepto la clase Object, extienden o heredan de una clase existente. En Java, la jerarquía de clases empieza con la clase Object, perteneciente al paquete java.lang, a partir de la cual heredan todas las clases en Java, ya sea en forma directa o indirecta. Si la declaración de una clase no especifica extends y el nombre de una clase a la derecha del nuevo nombre de la clase, esta nueva clase extiende implícitamente a la clase Object.

Los métodos con **modificadores** public, package, y protected podrán ser accesibles desde las subclases, mientras que los que tengan modificadores private no. Una subclase puede efectuar cambios de estado en los miembros private de la superclase, pero sólo a través de los métodos no privados que se proporcionen en la superclase y sean heredados por esa subclase.

En el caso de los **constructores de las subclases**, el constructor no sólo inicializa sus propios datos, sino que tiene que hacerlo también con los datos que proporciona la superclase. Además, primero se deben inicializar los valores de la superclase y luego los específicos de la subclase.

Para invocar desde el método constructor de la subclase al método constructor de la superclase se usa la palabra reservada **super** y entre paréntesis los parámetros del constructor de la superclase.

- La sentencia de llamada al constructor de la superclase debe aparecer siempre en primer lugar en el código de implementación del constructor de la subclase.
- Si no se utiliza en el constructor de la subclase una llamada explícita a uno de los constructores de la superclase, entonces se llamará automáticamente y de manera implícita al constructor por defecto de la superclase (constructor sin parámetros que el compilador crea de manera automática únicamente en el caso de que no se hubiera definido en la clase ningún otro constructor). Si la superclase tiene constructores deberá de hacerse explícitamente sino dará un error de compilación.

Para **redefinir un método** sólo tenemos que poner en el código de la clase un método con el mismo nombre y signatura que el método a redefinir, junto a su nueva implementación. Cuando desde una subclase se quiere invocar a un método redefinido de la superclase, se debe emplear la palabra clave super seguida de un punto y el nombre del método de la superclase al que se quiere invocar. Debemos tener en cuenta:

- Para redefinir un método de la superclase en una subclase, éste tiene que tener exactamente los mismos parámetros que en la superclase
- Cuando se redefine un método en una subclase, no se puede cambiar su modificador de acceso a uno más restrictivo que el que tenía en la superclase, aunque sí a uno menos restrictivo

La palabra clave **final** la podemos usar en diferentes contextos:

- Una variable de instancia puede ir precedida en su declaración del modificador final, queriendo esto decir que es una variable que debe ser inicializada en el método de creación del objeto y, además, que no podrá ser modificada en ningún otro método. Reciben también el nombre de **constantes blancas**. Por ejemplo para usar en identificadores únicos.

  [private | protected | public] final tipo_atributo nombre_atributo;

- Un método de una clase también puede ir acompañado del modificador final, queriendo esto decir que el método no puede ser redefinido en las subclases

  [private | protected | public] final tipo_retorno nombre_método(parámetros) excepciones;

- Un parámetro de un método puede ir acompañado del modificador final, queriendo esto decir que no se puede modificar su valor dentro del método.

  modificadores tipo_retorno nombre_método(final tipoParámetro nombreParámetro, ...) excepciones;

- Una clase puede ser definida como final. En este caso, el modificador impide que la clase en cuestión pueda ser superclase de ninguna otra

  [public] final class Nombre_de_la_clase [extends superclase] implements lista_de_interfaces

Polimorfismo

Polimorfismo es la propiedad de ciertos cuerpos de cambiar de forma sin variar su naturaleza. Podemos definir el polimorfismo como la posibilidad de que toda referencia a un objeto de una

superclase pueda tomar la forma de una referencia a un objeto de una subclase heredada de la anterior, lo cual nos va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía de clases, como si todos fueran objetos de sus superclases. Una variable puede tener un tipo definido en tiempo de compilación, pero luego, en tiempo de ejecución puede estar referenciando a objetos de tipos distintos, con la única condición de que estos tipos sean subclases directas o indirectas del tipo definido en tiempo de compilación para la variable.

Si contamos con una estructura de datos que contiene objetos de distintas clases se dice entonces que se trata de una **estructura de datos polimórfica o un contenedor polimórfico**. Para ello debemos hacer uso de características propias de la superclase y por lo tanto comunes.

La posibilidad de abstracción que nos abre el polimorfismo, al permitirnos poder programar hacia la generalidad en vez de programar para casos particulares y concretos, es lo que otorga a la programación orientada a objetos toda su potencia como filosofía de programación extensible.

Pero un objeto de la superclase no puede ser tratado como un objeto de la subclase, pues no soporta las novedades o especializaciones que aporta ésta. Así invocar un método exclusivo de una subclase sobre una variable del tipo de la superclase produce un error en tiempo de compilación.

Si una variable ha sido declarada con el tipo de la superclase, sólo se podrán invocar métodos propios de la superclase sobre el objeto al que hace referencia, aunque en tiempo de ejecución dicha variable esté haciendo referencia a un objeto de una de sus subclases.

Para poder acceder a métodos no comunes, en caso por ejemplo de una lista polimórfica, debemos hacerlo de la siguiente manera:

- Java me ofrece un operador lógico, llamado **instanceof**, que me permite saber si un objeto dado es de una clase determinada; es decir, que me permite identificar el tipo de un objeto en tiempo de ejecución. Por ejemplo:

  ```
  if (personaAModificar.dato instanceof TrabajadorFijo){
  }
  ```

- Para acceder a los métodos tenemos que hacer casting

  ```
  TrabajadorFijo unTrabajadorFijo = (TrabajadorFijo) personaAModificar.dato;
  ```

  sólo puede realizarse con éxito si la variable origen está referenciando en tiempo de ejecución a un objeto de la clase de la variable destino, pues, de otro modo, se produciría un error definido por la excepción ClassCastException.

Llamamos **ligadura dinámica** a la capacidad de los lenguajes orientados a objetos de determinar, en tiempo de ejecución, qué versión o implementación de un método tiene que ser ejecutada cuando dicho método ha sido redefinido en las subclases y, además, se invoca sobre una variable polimórfica.

Clases abstractas

Existen casos en los que es conveniente declarar clases para las cuales el programador no pretende instanciar objetos, sino simplemente utilizarlas como superclases en alguna jerarquía de herencia. Es posible que en una clase de este tipo, es decir, en una clase cuya única función es la de ser superclase en una jerarquía, haya métodos para los que no exista implementación.
Un método que no proporciona ninguna implementación recibe el nombre de método abstracto y

en su definición deberá aparecer la palabra clave abstract después del modificador de acceso del atributo y antes del tipo del valor de retorno del mismo:

[private | protected | public] abstract tipo_retorno nombre_método (parámetros) excepciones;

Una clase que contiene algún método que sea abstracto recibe, a su vez, el nombre de clase abstracta y en su definición deberá aparecer la palabra clave abstract después del modificador de acceso de la clase y antes de la palabra clave class:

[public] abstract class Nombre_de_la_clase [extends superclase] implements lista_de_interfaces

Debemos tener en cuenta:

- Una clase que contenga métodos abstractos debe estar definida obligatoriamente como clase abstracta.
- Las clases abstractas se utilizan sólo como superclases en las jerarquías de herencia y no se pueden utilizar para instanciar objetos. Si se intenta instancia un objeto de una clase abstracta se producirá un error en tiempo de compilación.
- Las subclases deberán dar implementación a los métodos abstractos y recibirán el nombre de **clases concretas**.
- Es posible que una subclase de una clase abstracta sea también abstracta pero las clases terminales de una jerarquía de herencia serán obligatoriamente clases concretas.
- Los métodos constructores no se heredan, por lo que no pueden declararse como abstract.

Interfaces

Una interfaz es una clase donde todos sus métodos son abstractos. Mediante la construcción de una interface, el programador pretende especificar qué comportamiento caracteriza a una colección de objetos e, igualmente, especificar qué comportamiento deben reunir los objetos que quieran entrar dentro de esa categoría o colección.

Una interface en Java es una declaración de métodos sin implementación, aunque también se pueden declarar constantes, que definen el comportamiento que deben soportar los objetos que quieran pertenecer al "club" de esa interface. La diferencia entre intefaz y clase abstracta es un poco más profunda:

- Una clase abstracta define una interfaz que sólo puede ser utilizada a través de la herencia. Utilizar este mecanismo implica establecer vínculos fuertes entre las clases

- Una interface puede ser implementada por cualquier clase. Clases que no tengan ninguna relación entre sí, entiéndase ninguna relación de herencia, pueden compartir una interfaz. Deberemos utilizar interfaces cuando queramos definir un comportamiento, y sólo comportamiento, que pueda ser común a clases entre las que no haya relaciones de herencia. Debemos tener siempre presente que lo único que puede definir una interface son métodos y constantes, una interface no puede definir variables de ningún tipo.

Para definir una interface en Java haremos uso de la siguiente sintaxis:

```
Modificador_acceso interface Nombre_Interface {
        // Declaración de los métodos que constituyen la interface y definen el
        // comportamiento común a las clases que vayan a implementarla
}
```

Las interfaces también llevan modificador de acceso, que puede ser public o puede no aparecer nada, siendo entonces, por defecto, considerada la interface como de ámbito package. Tendrán asociado un archivo propio con extensión .java y que tendrá el mismo nombre que la interface.

Dentro de la interface hay que declarar los métodos que la constituyen y que definirán el comportamiento representado por la misma. Para definirlos haremos uso de la misma sintaxis que usamos para cualquier otro método, con las siguientes particularidades:

- A los métodos no se les da implementación, sólo se define su interfaz. Por lo tanto, son lo que en Java llamamos métodos abstract, aunque no es necesario ponerlo expresamente.
- Todos los métodos serán públicos, por lo que realmente no hay que declararlos expresamente como tales, aunque sí conveniente.

Un ejemplo es:

```
public interface ReceptorDeMensajes {
   public void enviarMensaje(String subject, String cuerpo);
}
```

Una interface también puede definir constantes de la siguiente forma:

```
public static final tipoConstante nombreConstante = valorConstante;
```

Por su parte, cuando una clase quiere soportar el comportamiento definido por una interface, debe implementarla. Para ello, deberá indicarlo así en su cabecera de declaración de clase, haciendo uso de la palabra clave implements, seguida del nombre de la lista de interfaces que implementa:

```
[public][final | abstract] class Nombre_clase [extends superclase][implements
lista_Interfaces]
```

No debemos olvidar nunca lo siguiente:

- Una clase puede implementar varias interfaces, separándolas por comas
- Una clase tan sólo puede extender o heredar de una superclase.
- Una clase que implementa una interface debe de proporcionar implementación para todos y cada uno de los métodos definidos en la misma, pues a todos los efectos, esos métodos forman parte de la interfaz pública de la propia clase. Si para algún método no quisiese proporcionar implementación, entonces la clase tendría que ser declarada como abstracta, al tener métodos sin implementación. No hacerlo así produciría un error en tiempo de compilación al ser un error de sintaxis.
- Las clases que implementan una interfaz que tiene definidas constantes pueden usarlas en cualquier parte del código de la clase, simplemente con su nombre. Tambien así: NombreInterface.NombreConstante.

Se **puede hacer un uso polimórfico** en base a una interface, de tal manera que puedo tratar de la misma forma objetos distintos con la única condición de que todos ellos implementen la misma interface. Para poder hacer uso del polimorfismo con interfaces deben cumplirse básicamente tres condiciones:

- El tipo de la variable polimórfica debe ser la propia interface. Es decir, si la interface tiene el nombre de InterfaceA, la declaración de la variable se hará de la siguiente manera: InterfaceA variable;
- Todos los objetos a los que haga referencia la variable polimórfica deben implementar obligatoriamente la interface sobre la que está definida la variable.

- Sobre una variable polimórfica sólo se podrán invocar métodos que vengan definidos en la propia interface y ningún otro.

Podemos simular la herencia múltiple haciendo uso de interfaces aunque no será exactamente lo mismo. Como siempre que se hace uso de la herencia múltiple, aunque ésta sea simulada a través de interfaces, cabe la posibilidad que se produzcan dos problemas:

- Colisión de nombres: una clase implementa dos interfaces que tienen métodos que se llaman igual. Puede pasar:
    - Si ambos métodos tienen el mismo nombre pero diferentes parámetros, entonces se produce una sobrecarga de métodos
    - Si ambos métodos tienen el mismo nombre y los mismos parámetros, pero se diferencian en el tipo del valor devuelto, se producirá un error de compilación, al igual que sucedía con la sobrecarga común.
    - Si ambos métodos coinciden exactamente en su declaración: nombre, parámetros y tipo del valor devuelto; entonces se eliminará uno de los métodos, y sólo se podrá implementar uno de ellos.
- Herencia repetida: Una interface en Java también puede heredar de otra interface absorbiendo todas las definiciones de métodos y constantes de su interface superclase. Si dos interfaces heredan de la misma interfaz y luego se usan para una clase concreta se produce herencia repetida. En este caso, de cada pareja de métodos repetidos se eliminará uno de ellos y sólo se podrá implementar el otro.

UNIDAD 16: Entorno de desarrollo integrado (IDE) NetBeans

IDE de programación multiplataforma para JAVA

UNIDAD 17: Introducción a la creación de interfaces gráficas de usuario (GUI) mediante Swing

AWT, Swing y las JFC

JFC son las siglas de **Java Foundation Classes**, que constituye una colección bastante completa de librerías de clases que proporciona el lenguaje Java, de forma gratuita, para permitir al programador disponer de una enorme funcionalidad lista para usar. Entre las clases de las JFC se incluye un importante grupo de elementos que ayudan a la construcción de interfaces gráficas de usuario (GUI) en Java.

Los elementos que componen las JFC son:

- **Componentes Swing:** que comprende componentes tales como botones, cuadros de texto, ventanas o elementos de menú.
- **Soporte de diferentes aspectos y comportamientos (Look and Feel):** Permite la elección de diferentes apariencias de entorno. Por ejemplo, el mismo programa puede adquirir un aspecto Metal Java (multiplataforma, igual en cualquier entorno), Motif (el aspecto estándar para entornos Unix) o Windows (para entornos Windows).
- **Interfaz de programación Java 2D:** Permite a los programadores incorporar gráficos en dos dimensiones, texto e imágenes de alta calidad.
- **Soporte de arrastrar y soltar (Drag and Drop)** entre aplicaciones Java y aplicaciones nativas. Es decir, se implementa un portapapeles.
- **Soporte de impresión.**
- **Soporte de sonido:** Captura, reproducción y procesamiento de datos de audio y MIDI
- **Soporte de dispositivos de entrada** distintos del teclado, para japonés, chino, etc.
- **Soporte de funciones de Accesibilidad,** para crear interfaces para discapacitados: Permite el uso de tecnologías como los lectores de pantallas o las pantallas Braille adaptadas a las personas discapacitadas.

Es posible encontrar ejemplos de casi todos ellos (menos de los tres últimos) en los ejemplos que nos proporciona la web dedicada a ejemplos de Java Web Start. **Java Web Start** permite ejecutar aplicaciones de tipo "stand alone" de forma remota, facilitando la descarga e instalación, y haciéndolas transparentes al usuario, por lo que es una estupenda herramienta de distribución de aplicaciones Java.

AWT (Abstract Window Toolkit)

Conjunto de paquetes y clases destinadas a proporcionar un Kit de herramientas para crear ventanas abstractas que pudieran incluirse en las aplicaciones que se desarrollaran con el lenguaje.

- Los desarrolladores del AWT original usaban una ventana del sistema operativo para cada uno de los posibles componentes. De esta forma, cada cuadro de texto, cada casilla de verificación, cada botón, etc. tenía su propia ventana, que debía ser gestionada por el sistema operativo. Esto significaba que cualquier aplicación tenía que gestionar un enorme

número de ventanas, y por ello cualquier aplicación gráfica consumía una gran cantidad de recursos del sistema (memoria y tiempo de CPU).
- Las clases AWT estaban desarrolladas usando código nativo (es decir, código asociado a plataformas concretas). Eso dificultaba la portabilidad de las aplicaciones. Era necesario restringir la funcionalidad a los mínimos comunes a todas las plataformas dónde se pretendía usar AWT
- El modelo de programación de AWT inicial no era ni siquiera orientado a objetos (aunque sí lo es el actual, desde Java 1.1).
- AWT sigue siendo imprescindible, ya que todos los componentes Swing se construyen haciendo uso de clases de AWT.

Las clases asociadas a cada uno de los componentes AWT se encuentran en el paquete java.awt y las clases relacionadas con el manejo de eventos están en el paquete java.awt.event.

Netscape sacó una librería de clases llamada **Internet Foundation Classes** para usar con Java, y eso obligó a Sun a reaccionar para adaptar el lenguaje a las nuevas necesidades. Se desarrolló en colaboración con Netscape todo el conjunto de componentes Swing que se añadieron a la JFC. Por tanto, Swing es el fruto de la colaboración entre Netscape y Sun. Lista de las clases AWT más populares:

- Applet: Ventana para una applet que se incluye en una página web.
- Button: Crea un botón de acción.
- Canvas: Crea un área de trabajo en la que se puede dibujar. Es el único componente AWT que no tiene un equivalente Swing.
- Checkbox: Crea una casilla de verificación
- Label: Crea una etiqueta
- Menu: Crea un menú
- ComboBox: Crea una lista desplegable
- List: Crea un cuadro de lista.
- Frame: Crea un marco para las ventanas de aplicación.
- Dialog: Crea un cuadro de diálogo.
- Panel: Crea un área de trabajo que puede contener otros controles o componentes.
- PopupMenu: Crea un menú emergente.
- RadioButton: Crea un botón de radio.
- ScrollBar: Crea una barra de desplazamiento
- ScrollPane: Crea un cuadro de desplazamiento
- TextArea: Crea un área de texto de dos dimensiones.
- TextField: Crea un cuadro de texto de una dimensión.
- Window: Crea una ventana.

Swing

Podemos decir de Swing que:

- Es una librería de clases dirigidas al diseño de interfaces gráficas de usuario.
- Para ello las clases Swing implementan la funcionalidad básica, de una forma flexible
- Es independiente de la arquitectura (tecnología no nativa propia de Java)
- Proporciona todo lo necesario para la creación de entornos gráficos, tales como diseño de menús, botones, cuadros de texto, manipulación de eventos, etc.
- Por cada componente AWT (excepto Canvas) existe un componente Swing equivalente, cuyo nombre empieza por J, pero reescrito enteramente en Java, y que permite más funcionalidad siendo menos pesado.
- Swing aumenta el número de componentes que se pueden usar respecto a AWT

- Los componentes Swing no necesitan una ventana propia del sistema operativo cada uno, si no que son visualizados dentro de la ventana que los contiene mediante métodos gráficos, por lo que requieren bastantes menos recursos.
- Las clases Swing están completamente escritas en Java, con lo que la portabilidad es total, a la vez que no hay obligación de restringir la funcionalidad a los mínimos comunes de todas las plataformas
- Debido a sus características, los componentes AWT se llaman componentes "de peso pesado" por la gran cantidad de recursos del sistema que usan, y los componentes Swing se llaman componentes "de peso ligero"
- Aunque todos los componentes Swing derivan de componentes AWT y de hecho se pueden mezclar en una misma aplicación componentes de ambos tipos, se desaconseja hacerlo. Es preferible desarrollar aplicaciones enteramente Swing, que requieren menos recursos y son más portables.

Un paquete es una carpeta en la que se guardan una serie de clases que tienen alguna relación entre sí. Todas las clases Swing están recogidas dentro del paquete javax.swing. La lista completa es la siguiente:

- javax.swing
- javax.swing.border
- javax.swing.colorchooser
- javax.swing.event
- javax.swing.filechooser
- javax.swing.plaf
- javax.swing.plaf.basic
- javax.swing.plaf.metal
- javax.swing.plaf.multi
- javax.swing.table
- javax.swing.text
- javax.swing.text.html
- javax.swing.text.html.parser
- javax.swing.text.rtf
- javax.swing.tree
- javax.swing.undo

En nuestra aplicación tendremos que incluir al menos la sentencia:

import javax.swing.;*

Contenedores Swing

Swing proporciona dos tipos de elementos contenedores:

- **Contenedores de alto nivel.** (Marcos: JFrame y Jdialog para aplicaciones) También JApplet, para applet, pero por ahora nos centramos en las aplicaciones stand alone. La ventana principal o marco sería el contenedor de alto nivel de nuestra aplicación, toda aplicación tiene al menos una. Los contenedores de alto nivel son componentes "peso pesado" necesitan crear una ventana del sistema operativo independiente para cada uno de ellos. El resto de componentes, serán de "peso ligero", es decir, que no tienen su propia ventana del sistema operativo, si no que se dibujan en su objeto contenedor. Existen 4 contenedores de algo nivel:
    - **JFrame**: Crea un marco, una ventana principal para la aplicación, que consta de Barra de título, con su menú de control, los botones de minimizar, maximizar/retaurar y cerrar, y con los bordes que la delimitan.
    - **JDialog**: Crea una ventana secundaria, un cuadro de diálogo que se abre para

interactuar con el usuario
    - **JApplet**: Crea un marco para una applet, es decir, para la ejecución de una aplicación Java integrada en una página web HTML, en una ventana del navegador.
    - **JWindow**: es un contenedor que puede ser mostrado en cualquier parte del escritorio del usuario. No tiene ni la barra de título, ni ningún botón de manejo de ventana, ni ninguna funcionalidad asociada, pero es un elemento del escritorio del usuario

  JFrame, al igual que todos los otros contenedores de alto nivel, contiene un objeto **JRootPane** como su único componente hijo. Ese objeto es un panel encerrado en los límites de la ventana, y podríamos representarlo como el panel de contenido (content pane), es decir, en él se "colgarán" todos los componentes mostrados por la ventana JFrame, salvo los elementos del menú. Este será un contenedor de bajo nivel sobre el que colocar el resto de componentes de la ventana. Podemos añadir componentes usando el método add() y eliminarlos con el método remove(), y que ese panel de contenido tendrá establecido por defecto algún gestor de distribución (Layout Manager) que indicará donde situar en la ventana cada nuevo componente.

- **Contenedores de bajo nivel.** (Paneles: JRootPane , Jpanel) Los paneles no necesitan su propia ventana del sistema, por lo que los llamamos contenedores de bajo nivel. Un JRootPane está compuesto por:
    - **glassPane** (objeto de la clase Component) Es una especie de panel o capa transparente, que siempre se sitúa encima de todos los demás para interceptar los movimientos del ratón. Ocupa todo el área visible de la ventana.
    - **layeredPane** (objeto de la clase LayeredPane) Es el encargado de gestionar el panel de contenido y la barra de menú, en caso de que la aplicación tenga una barra de menú. Se sitúa detrás del glassPane, y sobre él estará la capa de menús, de forma que cuando se despliegue un menú se dibuje por encima del resto de componentes de la ventana. Ocupa también todo el área visible de la pantalla.
    - **menuBar** Es opcional, y puede no estar. Si la aplicación dispone de barra de menús, será un objeto de tipo JMenuBar. Podemos considerarlo como otra capa, que está por encima de la capa de contenidos de la ventana, y que por tanto, siempre se ve dibujada encima del resto de componentes de la ventana.
    - **contentPane** Es un objeto de la clase Component. Es sin duda el panel con el que más interactuaremos. Generalmente los controles y los gráficos se sitúan en el contentPane. Es también una parte del layeredPane, que ocupa todo el área visible de la ventana, a exepción de lo que ocupe menuBar, si existe. Desde el programa, podemos acceder al contentPane de una ventana que hayamos creado mediante el método getContentPane() disponible para las clases JFrame y Japplet. También es posible crear un panel, de cualquier tipo (basta con que sea una subclase de Component) lleno de componentes que queremos que tenga nuestra aplicación, y decirle mediante el método setContentPane() que queremos que ése sea el panel de contenido de nuestra ventana.

Arquitectura MVC (Modelo-Vista-Controlador)

La arquitectura MVC significa que:

- El modelo de un componente está donde están almacenados sus datos.
- La vista es la representación en pantalla del componente, es decir, es justamente el "dibujo" del componente que nosotros vemos en la ventana de la aplicación.
- El controlador es la parte del componente que gestiona los eventos, como los clicks del

ratón. Cada una de las interacciones posibles con el usuario de un componente es lo que denominamos evento.

Manejo del aspecto y comportamiento (LookAndFeel)

Existen 3 tipos de aspecto o apariencia (look and feel) proporcionados gratuitamente junto al JDK:

- Apariencia Metal. Es la apariencia típica de Java, independiente de la plataforma, o multiplataforma.
  javax.swing.plaf.metal.MetalLookAndFeel

- Apariencia Motif. Es la apariencia para plataformas Unix (por ejemplo, para el sistema operativo Linux).
  com.sun.java.swing.plaf.motif.MotifLookAndFeel

- Apariencia Windows. Es la apariencia para plataformas Windows.
  com.sun.java.swing.plaf.windows.WindowsLookAndFeel

Con Swing, puede establecerse cualquiera de esas apariencias, con independencia de la plataforma en la que nos encontremos. Para ello usaremos el método setLookAndFeel() de la clase UIManager, pasándole como argumento la cadena que indica el nombre de la clase que tiene la información de esa apariencia, indicando el paquete en el que está,

Veamos como:

- Para establecer el aspecto multiplataforma de Java, llamado aspecto Metal:
  UIManager.setLookAndFeel(javax.swing.plaf.metal.MetalLookAndFeel);

- Para establecer el aspecto de plataformas Unix, llamado Motif:
  UIManager.setLookAndFeel(com.sun.java.swing.plaf.motif.MotifLookAndFeel);

- Para establecer el aspecto de plataformas Windows

  UIManager.setLookAndFeel
  (com.sun.java.swing.plaf.windows.WindowsLookAndFeel);

También es posible establecer el aspecto multiplataforma y el aspecto del sistema en que se ejecute la aplicación usando métodos específicos que obtienen las clases adecuadas para pasárselas como parámetros al método setLookAndFeel() :

UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

El método getCrossPlatformLookAndFeelClassName() devuelve el nombre de la clase que implementa el aspecto multiplataforma por defecto, el aspecto Metal (o Java Look and Feel).

Ese método es getSystemLookAndFeelClassName(), y el nombre de clase que devuelve es el de la correspondiente al look and feel del sistema en el que se está ejecutando la aplicación.

Sun, además, ha creado un "look and feel" específico para plataformas Macintosh, pero este se suministra e instala por separado, no viene incluido en el JDK.

Cerrar la ventana y la aplicación

En Swing, las ventanas se cierran automáticamente con un icono en la esquina superior derecha que cierra la ventana de la aplicación, sin necesidad de que hagamos nada más. Además podemos:

- Podemos hacer que la ventana no esté visible, y sin embargo que ésta siga existiendo y ocupando memoria para todos sus componentes, usando el método setVisible(false).

- Podemos invocar para la ventana JFrame al método dispose(), heredado de la clase Window, que no requiere ningún argumento, y que borra todos los recursos de pantalla usados por esta ventana y por sus componentes, así como cualquier otra ventana que se haya abierto como hija de esta (dependiente de esta). Sin embargo, el objeto ventana sigue existiendo, y podría ser reconstruido invocando al método pack() o al método show(), aunque deberían construir de nuevo toda la ventana.

- Si queremos cerrar la aplicación, es decir, que no sólo se destruya la ventana en la que se mostraba, sino que se destruyan y liberen todos los recursos (memoria y CPU) que esa aplicación tenía reservados, tenemos que invocar al método System.exit()

Las ventanas JFrame de Swing permiten establecer una operación de cierre por defecto con el método setDefaultCloseOperation() definido en la clase Jframe. La método se le puede pasar un constante para realizan diferentes tareas, estas son:

- **DO_NOTHING_ON_CLOSE** (definida en WindowConstants con el valor 0): No hace nada; requiere que el programa maneje la operación en el método windowClosing() de un objeto WindowListener registrado para la ventana.
- **HIDE_ON_CLOSE** (definida en WindowConstants con el valor 1): Oculta automáticamente el marco o ventana después de invocar cualquier objeto WindowListener registrado.
- **DISPOSE_ON_CLOSE** (definida en WindowConstants con el valor 2): Oculta y termina (destruye) automáticamente el marco o ventana después de invocar cualquier objeto WindowListener registrado.
- **EXIT_ON_CLOSE** (definida en JFrame con el valor 3): Sale de la aplicación usando el método System.exit().Al estar definida en JFrame, sólo se puede usar con aplicaciones, no con applet.

El valor se fija por defecto a HIDE_ON_CLOSE. (Eso al menos es lo que dice la API de Java, pero por ejemplo, NetBeans lo fija por defecto a EXIT_ON_CLOSE).

Todas estas posibilidades sólo actúan sobre la ventana, pero si se quiere cerrar la aplicación, habrá que añadirle un **Listener** (un escuchador de eventos) para decirle de forma explícita que al cerrar la ventana

- Una forma sencilla de hacerlo es añadir ese escuchador mediante el uso de una clase Adapter (concretamente WindowAdapter)
- WindowAdapter nos proporciona una implementación del interface WindowListener, aunque con métodos vacíos, cuyo cuerpo no contiene ninguna sentencia.
- El interface WindowListener establece los métodos que debe incluir cualquier escuchador de eventos de ventana, y serán básicamente uno para cada evento posible. La lista de eventos posibles son:
    - abrir la ventana (windowOpened). Evento que se produce cuando la ventana se hace visible por primera vez, es decir, cuando se abre la ventana.
    - activar la ventana. (windowActivated). Evento que se produce cuando la ventana pasa a ser la ventana activa.

    - desactivar la ventana (windowDeactivated). Evento que se produce cuando la ventana deja de ser la ventana activa.
    - iconificar la ventana (windowIconified) Evento que se produce cuando la ventana pasa a estar minimizada.
    - deiconificar la ventana (windowDeiconified). Evento que se produce cuando la ventana deja de estar minimizada.
    - cerrar la ventana (windowClosing). Evento que se produce cuando la ventana ha sido cerrada por el usuario desde el menú.
    - ventana cerrada (windowClosed). Evento que se produce cuando la ventana ha sido cerrada usando dispose()
- Esto nos permite redefinir sólo el método correspondiente al evento que nos interesa capturar, y no todos los que establece el interfaceWindowListener, que son siete en total.

Un ejemplo:

```
Jframe f= new JFrame("Ejemplo de cierre de unaAplicación");

f.setBounds(100,100,300,300);
f.setVisible(true);

f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

f.addWindowListener(new WindowAdapter(){
    public void windowClosed(WindowEvent e){
        System.exit(0);
    }
});
```

UNIDAD 18: Introducción de interactividad. Componentes Swing básicos

Gestores de distribución (Layout)

La idea es que sean cuales sean esas características, el aspecto y la distribución de los componentes en nuestra ventana se adapten a las características de la ventana en el ordenador en que se ejecuta, de acuerdo a las reglas que hayamos establecido, para que la apariencia de nuestra aplicación siga siendo más o menos la deseada, o al menos, la mejor posible en cada circunstancia. Debemos hacer un buen diseño de la interfaz gráfica, en el que tengamos presentes las posibilidades a considerar, y debemos elegir el mejor gestor de distribución (Layout) para cada uno de los contenedores o paneles de nuestra ventana.

Esto se puede hacer mediante el método setLayout(), al que se le pasa como argumento un objeto del tipo de Layout que se quiere establecer.

FlowLayout

Situados uno detrás de otro, en hilera. El código sería este:

> *getContentPane().setLayout (new java.awt.FlowLayout(java.awt.FlowLayout.LEFT, 25, 25));*

Va a disponer los componentes alineados a la izquierda, y con un espaciado de 25 píxeles, tanto vertical como horizontal entre los componentes que se añadan, y entre estos y el borde del panel contenedor.

La **dirección** de la hilera queda determinada por la propiedad **componentOrientation** del contenedor:

- ComponentOrientation.LEFT_TO_RIGHT: Los componentes se van añadiendo empezando por la izquierda,
- ComponentOrientation.RIGHT_TO_LEFT: Los componentes se van añadiendo empezando por la derecha

Distribuye los botones horizontalmente hasta que no caben más botones en la misma línea, en cuyo caso se comienzan a añadir a otra nueva línea. La **alineación** de la línea queda determinada por la propiedad **align**. La alineación puede ser:

- LEFT: Cada línea de componentes debe ser alineada a la Izquierda
- CENTER: Cada línea de componentes debe ser centrada
- RIGHT: Cada línea de componentes debe ser alineada a la Derecha
- LEADING: Cada fila de componentes debe ser alineada al borde principal según el sentido de la orientación del contenedor. Por ejemplo, a la izquierda con orientación de izquierda a derecha (LEFT_TO_RIGHT)
- TRAILING: Cada fila de componentes debe ser alineada al borde posterior según el sentido de la orientación del contenedor. Por ejemplo, a la derecha con orientación de izquierda a derecha (LEFT_TO_RIGHT)

Las propiedades del entorno NetBeans:

- **Alignment (Left, Center o Right).** invocamos al método setAlignment (int align) de la clase FlowLayout, donde align es un valor de 0 a 4, o sus equivalentes constantes (LEFT, CENTER, RIGHT, LEADING, TRAILING). Estas dos últimas, no son muy usadas, y NetBeans no las incluye en el diseñador.

- **Horizontal Gap**. Establece la separación horizontal entre los distintos componentes. Realmente invoca al método setHgap (int hgap)

- **Vertical Gap.** Establece la separación vertical entre los distintos componentes Realmente invoca al método setVgap (int vgap)

BorderLayout

Divide el contenedor (normalmente este contenedor será un panel) en 5 zonas, de forma que al añadir un componente al contenedor con el método add(), como segundo argumento de este método podamos indicarle en qué zona del contenedor queremos situar ese componente. Esas 5 zonas posibles son: CENTER, NORTH, SOUTH, EAST y WEST

Si cambiamos el tamaño de la ventana, cambiará el tamaño de cada componente, pero no su posición relativa dentro de la ventana.

getContentPane().setLayout(new java.awt.BorderLayout(15, 15));

Los dos valores son hgap y vgap que también pueden ser fijados para el gestor de distribución mediante los métodos setHgap() y setVgap()

Es posible insertar varios componentes en una misma zona del contenedor, pero sólo será visible uno.

GridLayout

Distribuye los componentes de un contenedor en una rejilla o cuadrícula rectangular. El contenedor es dividido en rectángulos de igual tamaño, y cada componente se sitúa dentro de uno de esos rectángulos. Tiene un número de filas y de columnas.

Cuando el número de filas y columnas han sido fijados ambos a valores distintos de cero, ya sea por el constructor de GridLayout o bien por haber usado los métodos setRows() y setColumns(), el número de columnas se ignora. De hecho el número de columnas se calcula a partir del número de filas y del número de componentes que se han añadido realmente al contenedor gestionado por GridLayout.

Sólo tiene sentido especificar el número de columnas cuando el número de filas es cero. En ese caso, se distribuirán los componentes en el número de columnas especificado, usando tantas filas como sean necesarias para mostrar todos los componentes añadidos.

De entre los métodos disponibles en la clase GridLayout, además de los constructores, destacamos los siguientes:

- setHgap() Establece el espacio horizontal entre componentes.
- setVgap() Establece el espacio vertical entre componentes.
- setRows() Establece el número de filas para el GridLayout.
- setColumns() Establece el número de columnas para el GridLayout.
- getRows() Devuelve el número de filas que tiene el GridLayout.
- getColumns() Devuelve el número de columnas que tiene el GridLayout.

GridBagLayout

Con GridLayout todas las celdas de la cuadrícula tienen el mismo tamaño, y cada componente está en una sola celda, por lo que todos los componentes tienen el mismo tamaño, lo cual es una limitación. Esto se soluciona mediante el uso de GridBagLayout, que nos aporta mucha más flexibilidad. Características:

- GridBagLayout es un gestor de distribución flexible, que alinea componentes vertical y horizontalmente, sin requerir que sean del mismo tamaño. Cada objeto GridBagLayout mantiene una cuadrícula dinámica y rectangular de celdas, y cada componente ocupa una o más celdas, que conforman su área de visualización (display area).

- Cada componente manejado por GridBagLayout se asocia con una instancia de GridBagConstraints. El objeto constraints especifica dónde debe colocarse sobre la cuadrícula el área de visualización de un componente, y cómo el componente debe ser posicionado dentro de su área de visualización.

- GridBagLayout también considera el tamaño mínimo (minimum size) y el tamaño preferido (preferred size) del componente para determinar el tamaño del mismo.

- Para la orientación habitual del contenedor, que suponemos será left-to-right, la coordenada (0,0) representa la esquina superior izquierda del contenedor, con x incrementándose hacia la derecha, e y incrementándose hacia abajo.

Para usar un GridBagLayout con efectividad, debes personalizar uno o más de los objetos GridBagConstraints que están asociados con sus componentes, fijando una o más de las variables de instancia del GridBagConstrains, que te mostramos a continuación:

- **GridBagConstraints.gridx, GridBagConstraints.gridy**
  Especifica la celda que contiene la primera esquina del área de visualización del componente, donde la primera celda de la cuadrícula tiene como dirección gridx=0, gridy=0.
  Se usa GridBagConstraints.RELATIVE (el valor por defecto) para especificar que el componente será colocado inmediatamente a continuación (a lo largo del eje x para gridx o del eje y para gridy) del componente que fue añadido al contenedor justo antes que el componente en cuestión.

- **GridBagConstraints.gridwidth, GridBagConstraints.gridheight**
  Especifica el número de celdas en una fila (para la variable >gridwidth) o en una columna (para la variable gridheight) que ocupa el área de visualización de un componente. El valor por defecto es 1.
    - Se usa GridBagConstraints.REMAINDER para especificar que el área de visualización del componente ocupará desde la columna de comienzo del componente (gridx) hasta la última celda en la fila (para la variable gridwidth) o desde la fila de comienzo del componente (gridy) hasta la última celda en la columna (para la variable gridheight).
    - Se usa GridBagConstraints.RELATIVE para especificar que el área de visualización del componente ocupará desde la columna de comienzo del componente (gridx) hasta la primera celda ocupada por el siguiente componente en su fila (para la variable gridwidth) o desde la fila de comienzo del componente (gridy) hasta la primera celda ocupada por el siguiente componente en su columna (para la variable gridheight)

- **GridBagConstraints.fill**
  Se usa cuando el área de visualización del componente es mayor que el tamaño requerido

para determinar si (y como) redimensionar el componente. Valores posibles son:

- GridBagConstraints.NONE (por defecto)
- GridBagConstraints.HORIZONTAL (hace el componente suficientemente ancho para ajustar su área de visualización horizontalmente, pero no cambia su altura)
- GridBagConstraints.VERTICAL (hace el componente suficientemente alto para ajustar su área de visualización verticalmente, pero no cambia su anchura)
- GridBagConstraints.BOTH (hace que el componente se ajuste para ocupar su área de visualización completa.

- **GridBagConstraints.ipadx, GridBagConstraints.ipady**
  Especifica el relleno interno del componente dentro de la distribución, cuánto añadir al tamaño mínimo del componente. El ancho del componente será al menos su ancho mínimo más ipadx píxeles. Similarmente, la altura del componente será al menos su mínima altura más ipady.

- **GridBagConstraints.insets**
  Especifica el relleno externo del componente, el espacio mínimo entre el componente y los bordes de su área de visualización.

- **GridBagConstraints.anchor**
  Se usa cuando el componente es más pequeño que su área de visualización, para determinar dónde (dentro del área de visualización) se debe colocar el componente. Hay dos tipos de valores posibles: relativos y absolutos.
  Los **valores relativos** son interpretados como relativos a la propiedad ComponentOrientation del contenedor, mientras que los **valores absolutos** no. Son valores válidos:

  - Absolutos:
    - GridBagConstraints.NORTH
    - GridBagConstraints.SOUTH
    - GridBagConstraints.WEST
    - GridBagConstraints.EAST
    - GridBagConstraints.NORTHWEST
    - GridBagConstraints.NORTHEAST
    - GridBagConstraints.SOUTHWEST
    - GridBagConstraints.SOUTHEAST
    - GridBagConstraints.CENTER (por defecto)
  - Relativos:
    - GridBagConstraints.PAGE_START
    - GridBagConstraints.PAGE_END
    - GridBagConstraints.LINE_START
    - GridBagConstraints.LINE_END
    - GridBagConstraints.FIRST_LINE_START
    - GridBagConstraints.FIRST_LINE_END
    - GridBagConstraints.LAST_LINE_START
    - GridBagConstraints.LAST_LINE_END

- **GridBagConstraints.weightx, GridBagConstraints.weighty**
  Se usa para determinar cómo distribuir espacio, lo que es importante para especificar el comportamiento al redimensionar. A menos que se especifique un ancho para al menos un componente en una fila (weightx) y columna (weighty), todos los componentes se agrupan juntos en el centro del contenedor. Esto es porque cuando el ancho es cero (por defecto), el objeto GridBagLayout pone algún espacio extra entre su cuadrícula de celdas y los bordes del contenedor.

CardLayout

Es un gestor de distribución que trata cada componente en el contenedor como una carta. Sólo una carta es visible en cada momento, y el contenedor actúa como una "baraja de cartas". El orden de las "cartas" (componentes) viene determinado por el orden interno de los componentes del contenedor.

El método addLayoutComponent (java.awt.Component, java.lang.Object) puede usarse para asociar un identificador de tipo String con una carta dada, para acceder directamente a visualizar esa carta.

BoxLayout

Permite distribuir sobre él múltiples componentes, ya sea horizontal o verticalmente. Los componentes, una vez distribuidos sobre el gestor, quedarán fijados en su sitio y no se moverán de ahí, de tal manera que, por ejemplo, si redimensionamos el marco y lo hacemos más pequeño, es posible que algún componente no se vea completo, pues estos no van a ser bajados a otra línea automáticamente por el simple hecho de disminuir el tamaño de la ventana. Se puede considerar que BoxLayout es una versión más amplia de FlowLayout

El gestor de distribución BoxLayout está construido con un parámetro axis (eje) que especifica el tipo de distribución que se hará. Hay cuatro posibilidades:

- X_AXIS - Los componentes son distribuidos horizontalmente de izquierda a derecha.
- Y_AXIS - Los componentes son distribuidos verticalmente de arriba a abajo.
- LINE_AXIS - Los componentes son distribuidos de la misma forma que las palabras se colocan en una línea. Si la propiedad ComponentOrientation del contenedor es horizontal, los componentes se distribuyen horizontalmente, y en caso contrario se distribuyen verticalmente. Para orientaciones horizontales, si la propiedad ComponentOrientation del contenedor es de izquierda a derecha (left to right), los componentes se distribuyen de izquierda a derecha, y en caso contrario se distribuyen de derecha a izquierda.
- PAGE_AXIS- Los componentes se distribuyen como las líneas en una página. Si la propiedad ComponentOrientation del contenedor es horizontal, los componentes se distribuyen verticalmente, y en caso contrario se distribuyen horizontalmente. Para orientaciones horizontales, si la propiedad ComponentOrientation del contenedor es de izquierda a derecha (left to right), los componentes se distribuyen de izquierda a derecha, y en caso contrario se distribuyen de derecha a izquierda.

Para cualquiera de las cuatro posibilidades, los componentes se ordenan en el mismo orden en que fueron añadidos al contenedor. BoxLayout intenta ordenar los componentes respetando su ancho preferido (preferred width), para distribución horizontal, o su altura preferida (preferred height) para distribución vertical.

- Para distribución horizontal, si no todos los componentes en la fila tienen la misma altura, BoxLayout intenta hacerlos a todos tan altos como el más alto de los componentes.
- Si esto no es posible para un componente particular, entonces BoxLayout alinea este componente verticalmente, de acuerdo con la propiedad Yalignment del componente.
- Por defecto, un componente tiene la propiedad Yalignment al valor de 0.5, lo que significa que el centro vertical del componente tendría la misma coordenada Y que los centros verticales de otros componentes con la propiedad Yalignment al valor 0.5
- Similarmente, para una distribución vertical, BoxLayout intenta hacer todos los componentes en la columna tan anchos como el más ancho de los componentes.
- Si esto falla, los alinea horizontalmente de acuerdo a la propiedad Xalignment.
- Para la distribución PAGE_AXIS , la alineación horizontal se hace en base al borde principal del componente. En otras palabras, un valor de Xalignment de 0.0 representa el

borde izquierdo de un componente si el contenedor tiene la propiedad ComponentOrientation fijada como de izquierda a derecha (left to right) y representa el borde derecho del componente en caso contrario.
- Lo habitual en nuestro caso, será que la propiedad ComponentOrientation sea efectivamente left to right.

En vez de usar BoxLayout directamente, muchos programas usan la clase Box, que es un contenedor de peso ligero que usa BoxLayout como gestor de distribución y que además proporciona métodos que ayudan al programador a usar y manejar BoxLayout con más facilidad. Añadir componentes a múltiples objetos Box anidados es una poderosa forma de conseguir el ordenamiento que se desea.

Null Layout

Null Layout es la ausencia de un gestor de distribución.

- Se usa para diseñar ventanas sin ningún gestor de distribución.
- Como en el caso de AbsoluteLayout, es útil para hacer prototipos rápidos
- Pero no se recomienda para producir aplicaciones finales, ya que las localizaciones y dimensiones fijas de los componentes no cambiarían cuando el entorno cambiara, lo que pude afectar a la correcta visualización de la ventana.

AbsoluteLayout

La otra posibilidad, muy similar, es AbsoluteLayout, pero en este caso es un gestor de distribución propio del entorno NetBeans, que no es por tanto compatible con el estándar de Java

- Permite situar los componentes exactamente donde se desee dentro de la ventana.
- Permite moverlos arrastrándolos con el ratón en el diseñador del entorno
- Permite redimensionarlos arrastrando sus bordes con el ratón.
- Es particularmente útil para hacer prototipos rápidos, en los que no hay limitaciones formales ya que no tienes que preocuparte por tener en cuenta ninguna configuración concreta.
- Sin embargo no es recomendable para producir aplicaciones "finales", dado que las localizaciones y dimensiones fijas de los componentes no cambiarían con el entorno, pudiendo afectar negativamente al aspecto que presenta la aplicación.
- Además, para distribuir una aplicación con el gestor de distribución AbsoluteLayout , propio del IDE, debes incluir las clases AbsoluteLayout. Está en

  <dir-instal>/ide5/modules/ext/AbsoluteLayout.jar

Uso de Paneles combinados con "Layouts"

El elemento que nos permite salvar la mayoría de limitaciones que nos imponen los Layout son los Paneles (JPanel). Consideramos un panel como un componente que queremos añadir a un contenedor que tiene un gestor de distribución determinado. Un componente es a su vez un contenedor que tiene su propio gestor de distribución. Ese gestor de distribución del panel puede ser distinto al del contenedor principal. Cada panel a su vez puede contener como componentes a otros paneles

Cuadros de texto

Cualquier propiedad de ese componente suele tener un método set y un método get asociado, que permiten respectivamente establecer un valor para una propiedad o consultar el valor que tiene asignado. Las propiedades son: background, border, editable, font, foreground, HorizontalAlignment, text, toolTipText, enabled, maximumSize, minimumSize, preferredSize y opaque.

Existe un método setLabelFor(), para JLabel, cuya finalidad es establecer un vínculo entre una etiqueta y un cuadro de texto, de forma que se pueda usar para que se active el campo de texto a través del mnemónico de la etiqueta. Si por ejemplo, la etiqueta tiene asociado en la propiedad displayedMnemonic como mnemónico la letra b, que aparece subrayada en la etiqueta, para usar el campo de texto asociado a esa etiqueta podremos pulsar Alt+b, en vez de usar el ratón para acceder al campo de texto. El método setLabelFor() es usado para mejorar la accesibilidad de la aplicación.

Con **FormatNumber** podemos formatear los textos numéricos para los cuadros.

```
FormateadorNumerico = NumberFormat.getInstance();
formateadorNumerico.setMaximumFractionDigits(4);
formateadorNumerico.setMinimumFractionDigits(1);
cuadro.setText(""+ formateadorNumerico.format(x1) );
```

getInstance() un conjunto de características "Locale". Locale es una clase que establece características locales para un país o idioma, tales como el carácter a usar para coma decimal o para separación de miles, etc.

```
NumberFormat formateadorNumerico = NumberFormat.getInstance(Locale.FRENCH);
```

Otro método que merece la pena ser destacado es el método setParseIntegerOnly (boolean value) que establece que sólo va a formatear la parte entera del número

Etiquetas

Las propiedades son: background, border, displayedMnemonic, font, foreground, HorizontalAlignment, icon, labelFor, text, toolTipText, enabled, maximumSize, minimumSize, preferredSize y opaque.

Casillas de verificación, botones de radio y grupos de botones

Las casillas de verificación en Swing están implementadas para Java por la clase **JCheckBox**, y los botones de radio o de opción por la clase **JRadioButton**. Los grupos de botones, por la clase **JButtonGroup**.

Ambos tienen dos "estados", se marcan o desmarcan usando el método setSelected(boolean estado) que establece el valor para su propiedad selected, le podemos preguntar si están seleccionados o no mediante el método isSelected() y podemos asociar un icono distinto para el estado de seleccionado y el de no seleccionado.

Un objeto de tipo **JButtonGroup** no es un objeto con una representación visible. Al añadirlo al diseño del interfaz, de hecho se guarda dentro del grupo de "otros componentes" en el diagrama de estructura del IDE

Las propiedades son: buttongroup, mnemonic y selected.

buttonGroup1.add(jRadioButton1);

Botones de acción y botones On/Off

Los botones de Acción al ser pulsarlos realizan una acción, y para Swing, la clase que los implementa en Java es **JButton**. Hay un tipo especial de botones, que funcionan como interruptores de dos posiciones o estados (pulsados-on, no pulsados-off). Esos botones especiales reciben el nombre de botones on/off o más frecuentemente **JToggleButton**.

En realidad, tanto JCheckBox como JRadioButton son subclases de JtoggleButton. Para manejar un JToggleButton, dispondremos de los métodos isSelected() y setSelected().

Listas

Para las listas **JList**, un modelo separado (ListModel) representa los contenidos de la lista. Esto quiere decir que los datos de la lista se guardan en una estructura de datos independiente, que llamamos modelo de la lista. Es fácil mostrar en una lista los elementos de un array o vector de objetos, usando un constructor de JList que cree una instancia de ListModel automáticamente a partir de ese array.

JList no soporta desplazamiento vertical (scrolling) con barra de desplazamiento por sí mismo. Para disponer de esta útil posibilidad, debemos incluir el componente JList dentro de un JScrollPane, que sí facilita esta característica. Se puede hacer así:

JScrollPane panelDesplazamiento= new JscrollPane(listaDatos);

o

JScrollPane panelDesplazamiento = new JScrollPane();
panelDesplazamiento.getViewport().setView(listaDatos);

La propiedad **selectionMode** no da la posibilidad de seleccionar un solo elemento, o varios elementos simultáneamente. Los valores posibles para la propiedad selectionMode para cada una de esas opciones son las siguientes constantes de clase del interface ListSelectionModel:

- MULTIPLE_INTERVAL_SELECTION: Es el valor por defecto. Permite seleccionar múltiples intervalos, manteniendo pulsada la tecla CTRL, o la tecla de mayúsculas
- SINGLE_INTERVAL_SELECTION: Permite seleccionar un único intervalo, manteniendo pulsada la tecla mayúsculas mientras se selecciona el primer y último elemento del intervalo.
- SINGLE_SELECTION: Permite seleccionar cada vez un único elemento de la lista.

El método **setSelectedIndex** permite establecer el valor de la propiedad selectedIndex, es decir, cual es el índice del elemento seleccionado. (Selecciona el elemento del índice que se le pasa como argumento). **getSelectedIndex()** que nos permite saber cual es el índice del elemento seleccionado

getSelectecValue() devuelve el objeto seleccionado, de tipo Object, sobre el que tendremos que aplicar un casting explícito para obtener el elemento que realmente contiene la lista. **setSelectedValue()** que nos permite establecer cual es el elemento que va a estar seleccionado.

Para el caso de que se permitan selecciones múltiples, contamos con los métodos análogos a los anteriores:

- **setSelectedIndices(),** al que se le pasa como argumento un array de enteros que representa los índices a seleccionar.
- **getSelectedIndices(),** que devuelve un array de enteros que representa los índices de los elementos o ítems que en ese momento están seleccionados en el JList.
- **getSelectedValues(),** que devuelve un array de Object con los elementos seleccionados en ese momento en el JList.

El contenido de un JList puede ser dinámico, es decir, los elementos de la lista pueden cambiar de valor y el número de elementos en la lista puede cambiar también, después de que el JList ha sido creado.

Los cambios en el modelo del JList son observados por un escuchador de eventos o listener de tipo **swing.event.ListDataListener**, que tendremos que implementar, y que observa eventos de tipo swing.event.ListDataEvent, que identifica el rango de índices que han sido modificados, añadidos o borrados.

Es posible usar el método setPrototypeCellVAlue() para establecer un tamaño fijo para todas las celdas, de forma que no se tenga que calcular el tamaño de cada celda.

Listas desplegables

La lista desplegable es una mezcla entre un campo de texto editable y una lista. Si la propiedad editable es true podrás modificar el texto desde teclado.

La propiedades son: model, prototypeDisplayValue, selectedItem, selectedIndex, editable, Mét._remove..., itemCount, Mét._getItemAt, Mét._selectedObjects y Mét._actionPerformed.

Programación guiada por eventos

Cualquier hecho que ocurre mientras se ejecuta la aplicación. Normalmente consideramos como eventos cualquier interacción que realiza el usuario con la aplicación, como puede ser pulsar un botón con el ratón, hacer doble clic, pulsar y arrastrar, pulsar una combinación de teclas en el teclado, pasar el ratón por encima de un componente, salir el puntero de ratón de un componente, o abrir una ventana, etc.

Para cualquier componente gráfico en la ventana, habrá un conjunto limitado de eventos que nos interese controlar.

1. Asociamos un escuchador de eventos (Listener) apropiado para el tipo de evento que queremos comprobar.
2. El listener no es más que un programa que permanece activo, en ejecución en segundo plano, de forma paralela a nuestra aplicación, y que se encarga exclusivamente de comprobar si se ha producido sobre el componente al que se ha añadido algún evento del tipo que él sabe escuchar.
3. En el momento que el listener "escucha" o intercepta un evento de ese tipo, lo captura, y pasa el control del flujo de nuestra aplicación al Controlador del componente al que está asociado
4. El controlador no es más que el código que el programador ha escrito para que se ejecute cuando se produce exactamente ese evento sobre ese componente. Este código recibe del listener además del control, un parámetro que es un objeto Evento del tipo del evento escuchado por el listener.

Para cada tipo de evento existe un interface que especifica los métodos que hay que implementar para tratar ese evento. Para añadir, por ejemplo, un evento de acción a un botón tendremos que hacerlo mediante la invocación para el botón del método addActionListener()

Se sigue un **convenio** bastante recomendable **para nombrar a ese método**: El método controlador empieza con el nombre del componente al que se ha asociado el escuchador, seguido del nombre del método del interfaz que hay que implementar para ese tipo de evento.

Java permite una sintaxis especial, que nos permite hacer una declaración de las clases necesarias incrustadas en el código de la clase en la que se quiere insertar el listener, (por eso se llaman clases internas) y sin necesidad siquiera de darle nombre a esa clase por parte del usuario (por eso se llaman clases anónimas), aunque sí será nombrada por el compilador de forma automática, y generará el correspondiente archivo .class, son las **clases internas anónimas**.

```
jB_AsignarCategoria.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
                jB_AsignarCategoriaActionPerformed(evt);
        }
});


private void jB_AsignarCategoriaActionPerformed(java.awt.event.ActionEvent evt) {
        String nombreEmpleado= (String) jL_Empleados.getSelectedValue();
        String categoria= (String) jCB_Categorias.getSelectedItem();
        jL_Notificaciones.setText("Empleado: "+nombreEmpleado+". Categoría Profesional:
        "+    categoria);
}
```

La posibilidad de gestionar eventos de documento (DocumentEvent) para que las modificaciones sobre un texto tengan una respuesta inmediata y automática, sin necesidad de usar ningún botón de acción se consigue con **DocumentListener**.

Manejo básico de los eventos de ventana

Existe un tipo especial de eventos de ventana (WindowEvent) y de escuchadores apropiados para ellos (WindowListener). El interface WindowListener obliga a implementar siete métodos:

- **windowActivated(WindowEvent e):** Este método debe implementarse con el código a ejecutar cuando la ventana pasa a ser la ventana activa.
- **windowClosed(WindowEvent e):** Este método debe implementarse con el código a ejecutar cuando la ventana ha sido cerrada invocando al método dispose()
- **windowClosing(WindowEvent e):** Este método debe implementarse con el código a ejecutar cuando el usuario intenta cerrar la ventana desde el sistema de menús de la misma
- **windowDeactivated(WindowEvent e):** Este método debe implementarse con el código a ejecutar cuando la ventana deja de ser la ventana activa.
- **windowDeiconified(WindowEvent e)**: Este método debe implementarse con el código a ejecutar cuando la ventana deja de estar minimizada
- **windowIconified(WindowEvent e):** Este método debe implementarse con el código a ejecutar cuando la ventana pasa a estar minimizada
- **windowOpened(WindowEvent e):** Este método debe implementarse con el código a ejecutar cuando la ventana se hace visible por primera vez.

La forma de evitar implementar los métodos que no necesito del interface es usando las clases **Adapter**. La clase Adapter nos proporciona ya una implementación para todos los métodos del interface que queremos implementar. Los métodos implementados por las clases Adapter no tienen ninguna funcionalidad, salvo la de cumplir con el requisito de implementar todos los métodos del interface, evitándonos la tediosa tarea de tener que escribir también los métodos que no necesitamos y que no nos interesan, lo cual no es poco.

Existen numerosas clases Adapter, y tienen más utilidad mientras mayor es el número de métodos que obliga a implementar un interface. Así, por ejemplo, no existe una clase ActionAdapter para implementar el interface ActionListener, puesto que tiene un solo método, que seguro que es el que queremos implementar con nuestro propio código, así que la clase Adapter carece de sentido.

```
class VentanaPrincipal extends JFrame{

VentanaPrincipal miVentana = new VentanaPrincipal("Ventana principal de la aplicación");

 miVentana.addWindowListener( new WindowAdapter(){
     public void windowClosing(WindowEvent e)
          {
            System.exit(0);
          }
 });

}
```

Esto es un ejemplo de uso de ActionAdapter con clases internas anónimas

UNIDAD 19: Más componentes Swing

Creación de un menú

Existen distintos componentes para crear menús:

- **JMenuBar, JMenu, JmenuItem:** La barra de menú no es más que el contenedor donde colocar todos los menús que queremos que incluya nuestra aplicación. Es casi un convenio que el primero de esos menús sea el menú Archivo, que tiene opciones relacionadas con las operaciones sobre ficheros. Puede hacerse mediante el propio diseñador del IDE, o mediante el código, generando un método setJMenuBar(), que añade una barra de menú ya creada a la ventana Jframe.

```
jMenuBar1 = new javax.swing.JMenuBar()
jMenu1 = new javax.swing.JMenu()
jMenuItem1 = new javax.swing.JMenuItem()
jMenu1.setText("Menu")
jMenuItem1.setIcon(new javax.swing.ImageIcon(getClass().getResource("/ix.png")))
jMenuItem1.setText("Item")
jMenu1.add(jMenuItem1)
jMenuBar1.add(jMenu1)
setJMenuBar(jMenuBar1)
```

- **Casillas de verificación (JcheckBoxMenuItem):** son componentes JcheckBoxMenuItem

- **Botones de opción dentro de un menú (JRadioButtonMenuItem):** son componentes JRadioButtonMenuItem. Suelen usarse para seleccionar opciones que son excluyentes entre sí, de forma que sólo una de ellas puede estar activa en cada momento, y necesariamente una de ellas debe estar activa. Para ello tenemos que hacer que los distintos JRadioButtonMenuItem pertenezcan al mismo JButtonGroup

- **Separadores (JSeparator)**: un elemento separador, de tipo JSeparator, que nos dibuja una línea horizontal en el menú

- **Submenús (JMenu como ítem dentro de un Jmenu):** para incluir un menú como submenú de otro basta con incluir como ítem del menú, a un objeto que también sea un menú, es decir incluir dentro del JMenu un ítem de tipo JMenu.

También es importante que cualquier componente interactivo de la ventana (cuadro de texto, botón de acción, lista desplegable, etc) y en particular cualquier opción del menú pueda seleccionarse sin el ratón, haciendo uso exclusivamente del teclado. La solución es la inclusión en todos los elementos del menú de **aceleradores de teclado o atajos de teclado** y de **mnemónicos**. Un acelerador o atajo de teclado no es más que una combinación de teclas que se asocia a una opción del menú, de forma que pulsándola se consigue el mismo efecto que abriendo el menú y seleccionando esa opción.

- Esa combinación de teclas aparecerá escrita a la derecha de esta opción del menú, de forma que el usuario pueda tener conocimiento fácilmente de su existencia.
- Añadir un atajo de teclado se consigue mediante la propiedad accelerator en el diseñador, que genera un método setAccelerator() como el que sigue:

jMenuItem2.setAccelerator(javax.swing.KeyStroke.getKeyStroke(
java.awt.event.KeyEvent.VK_M, java.awt.event.InputEvent.CTRL_MASK));

- También se puede hacer mediante un **Mnemónico** que consiste en resaltar una tecla dentro de esa opción, mediante un subrayado, de forma que pulsando Alt+<mnemónico del menú> se abra el menú correspondiente, y volviendo a pulsar la letra correspondiente se seleccione la acción correspondiente a esa opción del menú.

jMenuItem2.setMnemonic('m');

Barras de herramientas: JToolBar

Normalmente será rentable repetir funcionalidades para aquellos aspectos de nuestra aplicación que se ejecuten con mucha frecuencia, garantizando un acceso más rápido que por medio de los menús. Resulta muy cómodo agruparlos todos en una barra de herramientas. En Java las barras de herramientas se añaden como objetos de la clase **JToolBar**, y haciendo uso del diseñador. Si queremos que los botones sólo contengan una imagen, basta con eliminarles el texto en la propiedad text y asociarles un icono con la propiedad icon desde el diseñador.

Las principales propiedades de una barra de herramientas JToolBar son:

- floatable: Indica que va a ser (o no) una barra flotante
- orientation: Si toma el valor 0, la barra estará orientada en horizontal. Si toma el valor 1, estará orientada en vertical.
- rollover: Indica si los botones de la barra van a tener (o no) bordes alrededor

Paneles múltiples con pestañas: JTabbedPane

Para crear pestañas debemos: Añadir como panel de fondo de nuestra ventana un JTabbedPane, es decir, un panel con pestañas, a ese panel añadiremos nuevos paneles de tipo JPanel, por cada panel que añadimos se crea una nueva pestaña en el JTabbedPane y esos nuevos paneles JPanel, son contenedores a los que podemos añadir cualquier otro componente.

Las principales propiedades de JTabbedPane son:

- **selectedIndex:** Nos permite especificar el índice de la pestaña que queremos que se muestre en primer plano, como activa por defecto, al mostrarse el JTabbedPane.
- **tabPlacement:** Nos permite escoger el lugar donde se mostrarán las pestañas para seleccionar los distintos paneles. Por defecto toma el valor TOP (arriba), tal y como se muestra en la imagen que hay a continuación, pero puede tomar también los valores BOTTOM (abajo), LEFT (izquierda) o RIGHT (derecha)
- Para cada uno de los paneles que se incluyen en el JTabbedPane:
    - Tab Title: Permite indicar el texto que queremos que aparezca escrito en la pestaña de ese panel
    - Tab Icon: Permite incluir un icono en la pestaña de ese panel
    - Tab ToolTip: Permite indicar el texto de ayuda que aparecerá al parar durante unos segundos el puntero del ratón sobre la pestaña de ese panel.

Inclusión de elementos gráficos decorativos en el diseño.

Podemos añadir iconos a los **botones y equiquetas**, para ello:

- Incluimos en la carpeta de fuentes del proyecto (carpeta src del proyecto) los ficheros de imagen. Al compilar el proyecto se va a hacer una copia de esos ficheros en la carpeta build\classes
- Seleccionamos el botón o la etiqueta
- Seleccionamos la propiedad icon en la ventana de propiedades del diseñador y añadimos la imagen

Los **paneles** no tienen propiedad icon, así que no es posible asociarles una imagen de fondo mediante esta propiedad. Lo que sí podemos hacer es:

1. Poner una etiqueta que ocupe todo el panel
2. Asignarle a esa etiqueta un icono o imagen.
3. Añadir más elementos a ese panel, de forma que queden situados por encima de la etiqueta, que de esta forma seguiría actuado de fondo.

Para que la etiqueta se muestre realmente como fondo, tiene que estar al final de la lista de componentes incluidos en el panel.

NOTA: Si quieres mover algún componente de lugar en el panel, no puedes pincharlo y arrastrarlo directamente, ya que lo que estarías seleccionando y moviendo sería la etiqueta del fondo. Tendrás que seleccionarlo en la ventana Inspector, y llevarlo debajo de la etiqueta, tal y como hemos visto con el botón en las imágenes anteriores.

Selección de ficheros para entrada/salida mediante FileChooser

Un JFileChooser no es más que un tipo especial de cuadro de diálogo que ya nos proporciona Java. Ver ejemplos.

Ventanas internas con JInternalFrame

JinternalFrame crea ventanas que se muestran siempre dentro del marco de nuestra ventana principal de la aplicación. Al igual que las ventanas de tipo JFrame, sólo que al añadir a nuestro proyecto la clase de tipo JInternalFrame,

Para poder insertar un JInternalFrame el JFrame necesita que la ventana tenga un panel especial, de tipo JDesktopPane. Se estable al JFrame con setContentPane(). Luego se añadirá cada JInternalFrame a objeto JDesktopPane.

Ver ejemplos.

Cuadros de diálogo

Es cualquier ventana que se abre desde una aplicación para interactuar con el usuario. Al igual que las ventanas internas, no son contenedores de alto nivel, es decir, no son independientes, sino que tienen que estar asociadas a un componente padre, normalmente a la ventana JFrame de la aplicación. Es una ventana de tipo JDialog

La clase JDialog es la clase base para construir cuadros de diálogo. Nos ofrece dos posibilidades:

- Crear cuadros de diálogo personalizados, que consiste en diseñar una ventana de tipo Jdialog
- Crear cuadros de diálogo prefabricados, con la clase JOptionPane.

JOptionPane parece una clase complicada, por la gran cantidad de métodos disponibles, pero en realidad permite mostrar cuadros de diálogo más o menos estándar con relativa facilidad, ya que su uso se limita a hacer una llamada en una sentencia de una línea a un método showXxxDialog(), donde Xxx representa el tipo de cuadro de diálogo que queremos mostrar. Las posibilidades son:

- **showConfirmDialog():** Plantea una pregunta, con opciones: Si, No, Cancelar.
- **showInputDialog():** Pide al usuario la introducción de algún valor.
- **showMessageDialog():** Informa al usuario mediante un mensaje de algo que ha ocurrido.
- **showOptionDialog():** Es un tipo que unifica a los tres anteriores.

Los cuadros de diálogo generados por la clase JOptionPane tienen la propiedad modal a true, lo que significa que mientras que el cuadro de diálogo esté abierto, no es posible interactuar con otras ventanas de la aplicación, incluida la principal.

Los parámetros posibles son:

- **parentComponent:** Define el componente que va a ser el padre de este cuadro de diálogo, normalmente será la propia ventana JFrame desde la que se abre. Puede tomar el valor null, en cuyo caso se usa un Frame por defecto, y el cuadro de diálogo se sitúa en el centro de la pantalla.
- **Message:** Un mensaje descriptivo que se escribirá en el cuadro de diálogo, para preguntar o informar de algo
- **messageType:** Define el estilo del mensaje. El gestor del Look And Feel puede modificar el aspecto del cuadro de diálogo y proporcionar un icono por defecto. Los valores posibles, proporcionados como constantes de la clase JOptionPane, son:
    - ERROR_MESSAGE
    - INFORMATION_MESSAGE
    - WARNING_MESSAGE
    - QUESTION_MESSAGE
    - PLAIN_MESSAGE
- **optionType:** Define el conjunto de botones que aparecerá en el fondo del cuadro de diálogo. Los valores posibles, definidos como constantes de la clase JOptionPane, son:
    - DEFAULT_OPTION
    - YES_NO_OPTION
    - YES_NO_CANCEL_OPTION
    - OK_CANCEL_OPTION
- **options:** Proporciona una descripción detallada del conjunto de botones que aparecerán en el fondo del cuadro de diálogo. El valor usual de este parámetro es un **array de** String, de forma que se crea un botón por cada elemento del array al que se le pone dentro el texto del String para identificarlo.
- **Icon:** Proporciona un icono decorativo para colocar en el cuadro de diálogo. Si no se indica lo contrario, se usará el icono por defecto
- **title**: Establece el título para la barra de título del cuadro de diálogo. Es un String.
- **InitialValue:** Establece el botón de que aparecerá seleccionado por defecto.

Algunos de los métodos ShowXxxDialog () devuelven un entero. Ese número entero devuelto indica el botón que se ha seleccionado para cerrar el cuadro de diálogo. Los valores posibles son:

- YES_OPTION Indica que se ha salido pulsando el botón Sí (Yes)
- NO_OPTION Indica que se ha salido pulsando el botón No
- CANCEL_OPTION Indica que se ha salido pulsando el botón Cancelar (Cancel)
- OK_OPTION Indica que se ha salido pulsando el botón Aceptar (Accept)
- CLOSED_OPTION Indica que se ha salido pulsando el botón Cerrar (Close)

Tablas (JTable)

En este caso, la tabla es la vista, pero el modelo es la estructura de datos que queremos representar. Para el modelo parece evidente que un array bidimensional, sería una estructura muy adecuada. La propiedad model de JTable, nos permite decirle al compilador de dónde debe sacar los datos para llenar la tabla.

Las principales propiedades que podemos modificar para JTable son las siguientes:

- model
- autoCreateColumnsFromModel: Permite actualizar automáticamente la tabla cuando se producen cambios en el modelo.
- autoResizeMode: Permite determinar el modo en el que las columnas de la tabla cambiarán su tamaño cuando sea necesario, por aumentar el número de columnas, o por disminuir el tamaño del contenedor de la tabla.
- cellSelectionEnabled: Indica si se pueden o no seleccionar celdas individuales.
- columnModel: Establece el objeto que gobernará la forma en que las columnas se muestran en la vista.
- columnSelectionAllowed: Indica si se puede o no seleccionar por columnas, es decir, varias celdas de una misma columna.
- intercellSpacing: Establece el espacio vertical y horizontal de separación entre las celdas de la tabla.
- rowSelectionAllowed: Indica si se pueden hacer selecciones por filas, es decir, seleccionar varias celdas de la misma fila.
- columnCout: Indica el número de columnas de la tabla.
- rowCount: Indica el número de filas de la tabla.

UNIDAD 20: Acceso a bases de datos con Java

Introducción

Java, como los demás lenguajes de programación, es una gran herramienta para manipular datos, pero en cambio no provee una manera simple y eficaz de almacenar y tratar grandes cantidades de datos. Por ello debe trabajar conjuntamente con otras aplicaciones que sí estén especializadas en esas tareas. Esas aplicaciones informáticas son los Sistemas Gestores de Bases de Datos (SGBD).

Bases de datos y Sistemas de Gestión de Bases de Datos

Una **base de datos** es un conjunto de datos que tienen relación entre sí y que están almacenados con una cierta estructura que permite su manipulación. Es decir, añadir nuevos datos, modificar los existentes, borrar datos o consultarlos.

Un **sistema de gestión de bases de datos (SGBD)** es una aplicación informática que permite realizar las operaciones anteriores sobre una base de datos, de una forma rápida y sistemática. No hay que confundir base de datos con gestor de base de datos.

Normalmente una base de datos pretende modelar una situación del mundo real para permitir su tratamiento informático. Esto es, reproducir las informaciones del contexto que se pretende tratar, así como su estructura. A esto se le llama **modelo de base de datos**. A lo largo del tiempo han sido propuestos diferentes modelos, los más importantes son:

- **Modelo jerárquico:** Utiliza un esquema de representación basado en estructuras de tipo árbol jerárquico. En la actualidad no se utiliza.
- **Modelo en red:** Es una mejora del anterior en el cual se establecen relaciones no necesariamente jerárquicas. Como el anterior hoy día está en desuso.
- **Modelo relacional:** Es el más utilizado en la actualidad, representa la información por medio del concepto matemático de relación.
- **Modelo orientado a objetos:** Es el más reciente y trata de representar la información por medio de objetos, lo que lo hace especialmente interesante en combinación con lenguajes de programación como Java.

Bases de datos relacionales

El modelo relacional utiliza el concepto matemático de relación, que está basado en la teoría de conjuntos, para representar la información y su estructura. Gráficamente podemos imaginar una relación como una tabla que está formada por filas y columnas. Cada fila almacena información sobre un elemento del contexto que se pretende modelizar. Las columnas de la tabla, también llamadas atributos, representan los datos que se conocen de ese elemento. Una base de datos relacional suele constar de varias tablas que están relacionadas entre sí. Las tablas estarían relacionadas por medio del campo.

Para interactuar con el SGBDR se utiliza el lenguaje **SQL (Structured Query Language).** SQL es un lenguaje no procedimental en el cual se le indica al SGBDR qué queremos obtener y no cómo hacerlo. El lenguaje SQL está compuesto por comandos o instrucciones, cláusulas, operadores y funciones. Estos elementos se combinan para manipular las bases de datos.

Los comandos SQL se pueden dividir en dos grandes grupos:

- Los que se utilizan para definir las estructuras de datos, llamados **comandos DDL** (Data Definition Language).

- Los que se utilizan para manipular las estructuras **llamados DML** (Data Manipulation Language).

MySQL es un producto que ha evolucionado mucho desde sus primeras versiones y que se utiliza ampliamente en la actualidad. Es un SGBD que sigue el modelo relacional, es decir estructura los datos en forma de tablas, y para realizar operaciones sobre los datos utiliza el lenguaje SQL.

Acceso a bases de datos desde Java

Nuestro objetivo es escribir programas Java que utilicen los datos que están almacenados en un SGBD y para ello debemos conocer que es JDBC. **JDBC** son las siglas de **Java DataBase Connectivity**. Es una especificación de un conjunto de clases y métodos de operación que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea.

Para ello, entre el programa Java y el SGBD se interpone un elemento llamado controlador (driver) JDBC. Este controlador es el que implementa la funcionalidad de todas las clases de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD. La misión del controlador será traducir comandos del API JDBC al protocolo nativo del SGBD. Por lo que se podría cambiar de SGBD y de controlador JDBC sin tener que cambiar el código del programa Java y este seguiría funcionando. Los controladores JDBC aceptan SQL como lenguaje de consulta y manipulación de datos

El API JDBC soporta dos modelos de acceso a datos:

- **Modelo de dos capas:** En este caso la conexión entre la aplicación Java y el SGBD se hace de forma directa, lo que quiere decir que el controlador JDBC debe residir en el sistema local donde se ejecute la aplicación Java. En cambio el SGBD puede estar en otra máquina conectada en red. Ésta es la configuración más sencilla y la que usaremos en nuestras prácticas.

- **Modelo de tres capas:** En este modelo de acceso a las bases de datos, las instrucciones SQL son enviadas a un servidor intermedio entre la máquina que ejecuta la aplicación Java y la máquina que ejecuta el SGBD. Es un modelo más complejo de implementar que el de 2 capas, pero tiene la ventaja de que el nivel intermedio mantiene en todo momento el control del tipo de operaciones que se realizan con la base de datos, y además, los controladores JDBC no tienen que residir en la máquina cliente, lo cual libera al usuario de la instalación de cualquier tipo de controlador.

Existen cuatro categorías de controladores JDBC, cada una de ellas se diferencia de las demás por su forma de operar.

- **Controlador tipo I, puente JDBC-ODBC:** Se trata de un tipo de controladores diseñado para aprovechar las conexiones existentes que utilicen tecnología ODBC. ODBC (Open Database Connectivity) es una tecnología de Microsoft que funciona con una filosofía similar a la de JDBC, de manera que permite acceder desde las aplicaciones a las bases de datos gestionadas por un SGBD cualquiera, siempre que se disponga de un controlador ODBC adecuado. La desventaja es que es una tecnología muy ligada a los sistemas operativos de Microsoft. Los controladores JDBC de tipo I permiten acceder desde Java a

ODBC y desde ahí al SGBD. La ventaja es disponer de forma inmediata de las conexiones establecidas ODBC y la desventajas es su lentitud al interponer un nuevo eslabón en la cadena entre Java y el SGBD y que limitan su uso a sistemas Microsoft

- **Controlador tipo II, API Nativo:** se basa en una librería escrita en código nativo para acceder a la base de datos. El controlador traduce las llamadas JDBC a llamadas al código de la librería nativa, ésta tiene que ser proporcionada por los desarrolladores del SGBD. La ventaja es que es más eficaz que el tipo I y la desventaja que sigue teniendo el problema de la portabilidad al depender del código nativo.
- **Controlador tipo III, JDBC-Net:** en este caso el controlador se comunica con un servidor intermedio que se encuentra entre el cliente y el SGBD. El servidor intermedio se encarga de traducir las llamadas al API JDBC al protocolo específico de la base de datos. Las ventajas es que no se requiere ningún tipo de código nativo en el cliente (garantiza la portabilidad), el controlador es tecnología 100% Java y podremos emplear un mismo controlador para comunicarnos con diferentes SGBD. La desventaja es que controlador complejo. Está reservado a arquitecturas sofisticadas donde incluso se utilicen varios SGBD al mismo tiempo.
- **Controlador tipo IV, Protocolo nativo:** traducen directamente las llamadas al API JDBC al protocolo nativo del SGBD. La ventaja es que son los controladores que tienen mejor rendimiento y también emplea tecnología 100% Java. La desventaja es que está más ligado al SGBD que empleemos que los controladores de tipo III JDBC-Net

Siempre preferiremos los controladores de tipo III o IV. Normalmente los fabricantes de SGBD proveen de controladores JDBC que podemos descargar de Internet. También existen controladores desarrollados por terceras partes que pueden ser de código libre o comerciales.

La instalación es muy sencilla: basta con copiar el fichero del controlador JDBC en el directorio ext de nuestra instalación del JDK.

El API de Java para acceso a bases de datos java.sql

El paquete java.sql es el API de Java para acceso a bases de datos. Deberemos importarlo en todas las aplicaciones Java que desarrollemos para acceder a bases de datos. Toda aplicación que acceda a una base de datos empleando el API JDBC debe realizar una serie de pasos:

1. **Registrar el controlador JDBC:** Deberemos consultar la documentación del controlador que vamos a utilizar para conocer el nombre de la clase que hay que emplear. En MySQL es "com.mysql.jdbc.Driver", es decir, la clase Driver que está en el paquete com.mysql.jdbc. La línea de código necesaria sería:

   Class.forName("com.mysql.jdbc.Driver");

2. **Establecer una conexión con la base de datos:** Para ello se utiliza la clase DriverManager y el método adecuado:

   static Connection getConnection(String url, String user,String password)

   Este método intenta establecer conexión con la base de datos que le indiquemos en el campo url empleando para ello el controlador que hemos registrado anteriormente. Si tiene éxito devuelve un objeto Connection que es el que utilizaremos para comunicar con la base de datos. La formación del url dependerá del controlador JDBC que se utilice y que habrá que consultar la documentación suministrada con él. Para MySQL es:

jdbd:mysql://<Servidor MySQL> : <puerto comunicación> / <base de datos>

Finalmente tendríamos algo así:

```
miConexion = DriverManager.getConnection
        ("jdbc:mysql://localhost:3306/biblioteca", "root","123456");
```

3. **Ejecutar las instrucciones SQL:** Para ejecutar las instrucciones SQL se utilizan los objetos de tipo Statement o PreparedStatement, dependiendo de si queremos ejecutar SQL o SQL precompilado. La diferencia entre los dos métodos es que en el primer caso se pasan al controlador JDBC las instrucciones SQL cómo un String, mientras que en el caso de utilizar un objeto PreparedStatement se puede parametrizar la instrucción SQL proporcionando más flexibilidad a la hora de programar. Se utilizan los métodos definidos en la interface Connection:

    - Para ejecutar la consulta se utiliza el método executeUpdate(String sql) para instrucciones tipo UPDATE, DELETE, INSERT o instrucciones de tipo DDL, que son las que de alguna forma modifican la base de datos.
    - Para las instrucciones de tipo SELECT, que obtienen datos de la base de datos, pero que no la modifican, empleamos el método ResulSet executeQuery(String sql).

4. **Si las instrucciones SQL devuelven datos procesar los datos devueltos:** utilizando el método executeQuery() se obtiene un objeto de tipo ResultSet. Este objeto tiene una serie de métodos que sirven para recorrerlo y para extraer los datos que ha generado la ejecución del SELECT. Para obtener los datos de los atributos de cada fila del ResultSet se tienen los métodos getXXX(<nombre atributo>), donde XXX es el tipo de dato que se quiere utilizar.

    Para recorrer un ResultSet el método más utilizado es next(). Este avanza una fila en el ResultSet y devuelve un valor lógico verdadero (true) o falso (false) indicando si se ha llegado al final del ResultSet o no. Los demás métodos son:

    - boolean absolute(int row): Sitúa el cursor en la fila cuyo número se especifique como parámetro
    - void afterLast(): Sitúa el cursor al final del ResulSet, justo detrás de la última fila de datos.
    - void beforeFirst(): Sitúa el cursor al principio del ResulSet, justo delante de la primera fila de datos.
    - void close(): Cierra y libera todos los recursos usados por el ResulSet.
    - boolean first(): Mueve el cursor a la primera fila de datos del ResulSet, si existe una primera fila a la que moverse, devolviendo verdadero. Si no devuelve falso.
    - boolean isAfterLast(): Devuelve verdadero si el puntero está al final del ResulSet, detrás de la última fila de datos.
    - boolean isBeforeFirst(): Devuelve verdadero si el puntero está al principio del ResulSet, antes de la priemera fila de datos.
    - boolean isFirst(): Devuelve verdadero si el puntero está en la primera fila de datos.
    - boolean isLast(): Devuelve verdadero si el puntero está en la última fila de datos.
    - boolean last(): Mueve el puntero a la última fila de datos, y devuelve verdadero si ha sido posible. Devolverá falso si el ResulSet está vacío y no hay una última fila de datos a la que moverse.
    - boolean next(): Mueve el puntero a la fila posterior en el ResultSet, si existe una siguiente fila, devolviendo verdadero. Si no es posible por no existir dicha fila

(estamos ya al final del ResulSet), devolverá falso.

- boolean previous(): Mueve el puntero a la fila anterior en el ResultSet, si existe una fila anterior, devolviendo verdadero. Si no es posible por no existir dicha fila, (estamos ya al principio del ResulSet) devolverá falso.
- int getRow(): Devuelve el número de fila en la que está situado el puntero. La primera fila es la 1, la segunda es la 2 y así sucesivamente. Si no hay ninguna fila en el ResultSet, devuelve el valor cero.
- boolean Relative(int rows): Mueve el puntero el número de filas especificado como parámetro desde la posición de la fila actual. Si es positivo, hacia delante, y si es negativo hacia atrás. Devuelve verdadero si todo ha ido bien, y falso si nos hemos salido del ResulSet.

5. **Por último hay que liberar los recursos de la conexión:** Una vez que hayamos terminado de usar la conexión a la base de datos, hay que liberar los recursos que se estaban consumiendo en dicha conexión. Hay que liberar el ResultSet si lo hubiéramos utilizado, el objeto Statement y el objeto Connection. Todos estos objetos disponen de un método close() para liberar los recursos consumidos.

Cuando se produce un error se lanza una excepción del tipo java.sql.SQLException.

- Es importante que todas las operaciones de acceso a base de datos estén encerradas en un bloque try-catch que gestione las excepciones.

- Los objetos del tipo SQLException tienen dos métodos muy útiles para obtener el código del error producido y el mensaje descriptivo del mismo, getErrorCode() y getMessage() respectivamente.