1. Caso

Caso

*En cualquier empresa dedicada al desarrollo del software se precisa que los programadores sean capaces de diseñar por sí mismos las **rutinas** que después van a constituir la aplicación requerida. En **SI Andalucía** no son menos y están preparando a **Víctor** y a **Carmen** para que sean capaces de elaborar programas como miembros de un equipo de programación.*
*Parece que **Carmen** se desenvuelve muy bien y sólo tiene que adaptarse un poco a los métodos de trabajo de la empresa, viene bien preparada de sus estudios de Ciclo Formativo. Por su parte, **Víctor** tiene que aprenderlo todo, no sólo trabajar en equipo, sino también la programación en Java. Llevan unas semanas intentando iniciarlo en los aspectos básicos de un programador y aunque es un proceso lento la verdad es que el chaval está respondiendo muy bien haciendo todo cuanto le sugieren los compañeros. Se le nota que tiene ganas de aprender y a veces eso es lo más importante.*
*Cuando **José** comenzó explicándole el ABC de la programación, lo hizo comentándole la existencia de las **estructuras de datos** y cómo eran utilizadas mediante los algoritmos que posteriormente serían codificados a un lenguaje de programación. En su caso sería **Java**. Ahora que conoce el entorno de desarrollo NetBeans y ha desarrollado sus primeros programas de ejemplo en Java, ha llegado el momento de programar como lo haría un profesional. Es necesario elaborar los **programas** necesarios que van a completar la **aplicación informática** a la que se dedica el equipo y para ello es imprescindible en primer lugar que utilice correctamente las estructuras de datos estáticas, algo que en adelante va a aparecer en el noventa por ciento de los programas que desarrollen.*
***Víctor** recuerda que (según los apuntes que **Carmen** le dejó de ciclo formativo) en la unidad 2 sobre los tipos de datos y sus características, **José** dividía los datos en:*

- *primitivos*
- *y estructurados.*

*Ahora empieza a entender la importancia de este tipo de **datos** que hasta ahora ha utilizado en todos los programas que ha diseñado. Pero advierte que necesita saber cómo definirlos y utilizarlos en Java, probablemente sea suficiente ver algunos ejemplos de cómo se emplean y después intentar hacer un programa por sí mismo, bajo la supervisión de sus compañeros.*
*Tiene claro la gestión de memoria que hace el compilador al definir datos estáticos y cuando es conveniente su uso frente a las estructuras dinámicas, y eso según **Carmen**, ya es mucho.*

2. Introducción

Introducción

En las unidades anteriores hemos venido hablando de las estructuras de datos, de una forma más o menos general, o refiriéndonos a los datos de tipos primitivos. Ha llegado la hora de sumergirnos en las **estructuras de datos** que más suelen usarse en cualquier aplicación. El enfoque, como ya viene siendo habitual, es introducir las estructuras de datos tal y como funcionan en Java. Empezaremos distinguiendo entre dos grandes grupos de estructuras de datos:

- **Estructuras de datos estáticas**, de las que nos ocuparemos en esta unidad.
- **Estructuras de datos dinámicas**, de las que nos ocuparemos en la unidad siguiente.
- **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.

Si se necesita meter más datos en la estructura y ya está llena, no resulta posible. Habrá que crear otra estructura similar, pero más grande, y copiar en ella los datos de la antigua para seguir trabajando. Pero ya no será la misma estructura, si no una copia.

Por ejemplo, una cadena de caracteres se crea con una longitud. Si deseo almacenar en ella más letras, no puedo. La única posible solución a esto es que tengo que crear una nueva cadena de caracteres más larga, en la que copio la anterior, y le añado las nuevas letras. Básicamente, necesito buscar espacio en memoria para toda la nueva estructura cada vez que necesito que cambie su tamaño.

Otro **ejemplo** sería una tabla con los nombres de los alumnos de un curso. Si he previsto que la tabla tenga

30 filas, para un curso de 30 alumnos, y en un momento se matriculan 5 nuevos alumnos, no podré almacenarlos en esa tabla, no caben. Debo desechar esa tabla y construir una nueva, de 35 filas, en la que las 30 primeras serán copias de los valores de los 30 alumnos que ya estaban matriculados, y las otras 5 contendrán a los nuevos alumnos.

¿Y por qué no me curo en salud, y creo directamente una tabla de 40 alumnos, y así me caben todos?

Por dos motivos:

- **Estaría desperdiciando espacio**, ya que sé que es poco probable que necesite espacio para más de 30 alumnos. Recuerda, la memoria es siempre un bien escaso en programación, que hay que administrar lo mejor que se pueda.
- **Es posible que no esté interesado en imponer un tamaño máximo**, o mínimo, como en el caso de las cadenas de caracteres. **Incluso con una tabla de 40 filas, me puedo encontrar con un curso de 60 alumnos**, como es el caso de la FP a distancia, y **sigo teniendo el mismo problema**.

¿Por qué limitar el **tamaño máximo** en caracteres de una frase o de un párrafo?

Sólo en aplicaciones para los móviles, o algo similar, con escasa memoria, y donde no te sueles dedicar a escribir novelas, puede tener sentido.

¿Y qué pasa con la imposición de un **tamaño mínimo**?

Tampoco parece oportuno limitar el tamaño de cualquier cadena de caracteres a pongamos 2000 caracteres, para permitir mensajes largos. Estaría desperdiciando muchísima memoria para mensajes cortos, que probablemente son la mayoría.

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.

Es el caso, como vimos, de las **listas enlazadas**. Cada elemento de la lista, anota la dirección de memoria en la que está el siguiente, por lo que **no hay que reservar espacio** para elementos que no existen. Cuando se crea un nuevo elemento, basta con que el anterior anote y "recuerde" la posición de memoria en la que se ha creado, para que podamos recorrer toda la lista hasta llegar a él.

En la siguiente unidad, tendrás ocasión de ampliar los conocimientos sobre estructuras dinámicas de datos.

Por ahora, vamos a comenzar introduciendo algunos conceptos que nos pueden resultar necesarios para nuestros ejemplos.

3. Conceptos previos: modificador static

Conceptos previos: modificador static

*CASO. **Carmen** explica a **Víctor** que los programas en Java están formados por métodos que se ejecutan de forma independiente. Cada uno de estos métodos hay que verlo como un proceso que:*

- *recibe datos (parámetros)*
- *y que tras su ejecución devuelve un resultado final para el usuario o que se empleará como entrada en otro método.*

*Pero hay ocasiones en que no se emplean los métodos de modo independiente, sino que pueden **compartir datos**, de modo que una variable definida en un método es necesario utilizarla en otros*

métodos durante la ejecución del programa.
***Carmen** comenta que para ello Java utiliza durante la definición de las variables el modificador **static,** mediante el cual al concluir la ejecución de un método no se libera el espacio de memoria de la variable sino que es conocido por todos los métodos, incluso de otras clases. Para que **Víctor** lo entienda bien, decide ponerle un ejemplo.*

Las variables que definimos en una parte de un programa, por ejemplo dentro de un método, ¿son entendidas y utilizables fuera de ese método?

Lo normal es que no, pero si lo deseo**, 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.**

Existen situaciones en las que me interesa que una **variable sea global** para la clase, y en ese caso la defino con el modificador **static** delante de la declaración de la variable.

Ejemplo:

Si para la clase **`Empleado`** de nuestra empresa disponemos de métodos que requieren saber cuantos trabajadores hay contratados en cada momento, y que modifican cada uno de ellos ese valor, no podremos definir la variable **`totalTrabajadores`** dentro de los métodos, ya que una vez que termina la ejecución de un método desaparecen de la memoria todas las variables que había definido.

- Por ejemplo, el método **`contratarTrabajador()`** debe sumar uno a **`totalTrabajadores`** cada vez que se contrata a un nuevo trabajador,
- y el método **`darDeBajaTrabajador()`** debe restar uno a esa variable cada vez que se despide a un trabajador.
- Ambos modifican el valor, y ambos deben hacer que esas modificaciones sean permanentes, es decir, consultables incluso después de que haya terminado el método.
- De este modo el valor de la variable **`totalTrabajadores`** se perdería, y no podríamos ver desde otro método las modificaciones que se han hecho sobre él.
- Por ello debo definirla como **`static int totalTrabajadores`**, al principio de la clase, asegurando de esta forma que podrá consultarse y actualizarse su valor desde cualquier parte, ya que es una variable global para toda la clase.

Otro ejemplo similar lo tienes en la declaración de la variable **`private static int totalDepositosCreados=0;`** en la clase **`Deposito`** de los ejemplos de las unidades anteriores.

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. Por ejemplo, los métodos de la clase **`ES`** son todos **`static`**,ya que no hacen nada con ningún objeto **`ES`**. De hecho, no se van a crear objetos de la clase **`ES`**. Están ahí sólo para agruparlos en un módulo funcional de operaciones de entrada de datos desde teclado, y se ejecutarán con el nombre de la clase, desde otras clases.

Ejemplo: el método de lectura de un número entero se define como static en la clase **`ES`**...

```
public static int leeNº(String mensaje) {...}
```

...y se ejecuta desde la clase **`GestionDeposito`** como...

```
cantidad = ES.leeNº("¿Que cantidad desea sacar?");
```

Los métodos **no static** siempre se ejecutarán para un objeto, al que se le envían a modo de mensajes para que los interprete y los ejecute. Por ejemplo, el método **`public void vaciarDeposito(){...}`** de la clase Deposito, que no es estático, se ejecutará mediante un objeto, porque no hace algo general para toda la clase, sino que hace algo con un objeto concreto, a través del que se llama. Así, en la clase **`GestionDepositos`** tenemos una llamada como **`d1.vaciarDeposito();`** donde **`d1`** es el objeto al que se aplica el método.

El modificador **static** también significa "propio de la clase, y no de cada objeto", es decir, declara una variable

como un miembro de clase y no como un miembro de instancia u objeto. Pero eso lo dejamos para cuando comencemos con los conceptos de programación orientada a objetos, en unidades venideras. Por ahora sólo se menciona a modo de anticipación.

Conceptos previos: Llamadas a métodos

*Continuando con su exposición **Carmen** explica que los métodos son la base de la programación en Java, igual que lo son las funciones en C. Básicamente se trata de un **grupo de instrucciones ordenadas**, que se ejecuta para llevar a cabo una **tarea concreta** y al que se le da un **nombre** para invocarlo, cuando sea necesario desde cualquier parte del programa. Además dice que una de las cosas que debe dominar son los valores devueltos como resultado de ejecutar un método.*
***Víctor** dice que lo entiende y, buscando como siempre el lado práctico de las cosas, añade que sería como evitar reescribir un trozo del código cada vez que se quiera utilizar en un programa. Y para estar más seguro le pide a su compañera que le ponga un ejemplo.*

Una parte importante de las estructuras de datos es **aprender a hacer cosas útiles** con ellas, manipularlas para almacenar información, y esto resultará mucho más sencillo si antes te explicamos algunos conceptos sobre métodos que nos permitan usarlos con normalidad y con conocimiento de causa, en los programas que gestionan estructuras de datos.

Aunque ya han venido apareciendo métodos en los ejemplos vistos con anterioridad, ya que no es posible hacer nada en Java sin usar al menos el método `main()`, aún quedan algunos conceptos pendientes, y vamos a empezar a verlos desde el principio.

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.**

Así, por ejemplo, si un conjunto de sentencias más o menos significativo quiero ejecutarlas frecuentemente en un programa, lo mejor es **agrupar** todas esas sentencias en un **método** y llamar al método desde cualquier parte que necesite la ejecución de las sentencias que contiene.

Pero además, puedo mejorar la versatilidad de mi programa, si para cada ejecución puedo indicar de alguna forma que tenga en cuenta algunos datos en la ejecución de esas sentencias, de manera que

- **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.

Ejemplo. Para la clase `Empleado`, es posible disponer de un método que al ejecutarse nos devuelva el número de trabajadores de una localidad concreta (que pasaremos como parámetro), y eso podemos usarlo para calcular el porcentaje que corresponde sobre el total de la plantilla. Así para ver el tanto por ciento de trabajadores que la empresa tiene de Aguadulce podemos escribir una sentencia similar a la siguiente:

```
...
porcentajePorPoblacion = cuantosTrabajadores("Aguadulce") * 100 / totalTrabajadores;
System.out.println("De Aguadulce hay " + cuantosTrabajadores("Aguadulce") + " trabajador
System.out.println("Que corresponden al " + porcentajePorPoblacion + "% del total.");
...
```

`cuantosTrabajadores("Aguadulce")`, este método devuelve un entero con el total de trabajadores de esa población. A continuación escribimos un esquema de la posible declaración del método.

```
public int cuantosTrabajadores(String poblacion){

     // Aquí pondríamos el código necesario para contar los trabajadores que son de la
     // población pasada como parámetro. Usamos la variable nTrabajadores para sumarle
     // uno cada vez que se encuentre un nuevo trabajador de esa localidad.
     // La inicializamos a cero porque antes de empezar, aún no hemos encontrado ningún
     // trabajador de esa población.

    int  nTrabajadores=0;

    //(código necesario para recorrer todos los trabajadores. Dependerá de dónde estén
    // almacenados los datos, y básicamente será un bucle que se irá repitiendo para cada
    // trabajador, pasando a comprobar el siguiente, y mientras que queden trabajadores. )

    // Para cada trabajador comprobamos su población. Si coincide con la indicada como
    // parámetro, le sumaríamos uno al número de trabajadores. Naturalmente trabajador
    // es una referencia de la clase Empleado, que irá apuntando alternativamente a cada
    // uno de los trabajadores de la empresa.

    if (trabajador.poblacion = poblacion){
         nTrabajadores++;

    ...

    // Y para terminar devolvemos el valor calculado.

    return nTrabajadores;
}
```

Una cosa sí debemos tener presente:

> **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. Por tanto, para los tipos primitivos, las modificaciones sobre ellos no se ven desde fuera del método. Si alguna de ellas tiene información que necesito, debo declararla como variable global, o devolverla como valor de retorno del método.

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**.

Autoevaluación

4.1. Tipo devuelto por un método

Tipo devuelto por un método

En la definición de cualquier **método** siempre hay que indicar el **tipo** que devuelve.

¿Siempre? Pero es posible que yo no quiera o no necesite que mi método devuelva nada. Mi método debe hacer alguna tarea, por ejemplo escribir un mensaje en pantalla, pero no lo voy a incluir en ninguna expresión, por lo que devolver un valor no aporta nada, ya que no se recoge en ninguna variable, ni se usa para hacer ningún cálculo. ¿También tengo que especificar el tipo devuelto?

También, sólo que **si no deseo un tipo devuelto debo indicar que el método devuelve el tipo "vacío". Esto se indica con la palabra reservada `void`**.

En el ejemplo de la clase **`Deposito`** entre otros tenemos el método

```
public void vaciarDeposito(){..}
```

Este método se declara como void porque lo único que necesito es que se realicen las acciones necesarias para que un depósito se quede vacío, y que se indican en el cuerpo del método. No parece probable que necesite incluir este método en una expresión, como si fuera una función, ni nada parecido. No necesito por tanto ningún valor de retorno.

Sin embargo, el siguiente método sí devuelve un valor de tipo **boolean** (lógico). Debo indicar el tipo devuelto delante del nombre del método y recordar que en el cuerpo del método tendré que incluir una sentencia **`return`** seguida de una expresión o variable del mismo tipo.

Por ejemplo, cuando intento sacar una cantidad de un depósito, no siempre será posible. Sólo podré terminar correctamente la operación si la cantidad que contiene el depósito es suficiente. Si no lo es, anularé la operación.

```
public boolean sacarDeDeposito(int cantidad){
     boolean correcto=false;
     ...;
     return correcto;
}
```

Parece interesante que cuando llamo a este método sea posible comprobar si se ha realizado la operación o no. Ése es el sentido de devolver **`boolean`**.

- Si la operación fue bien, se devolverá **`true`**,
- y en caso contrario **`false`**.
- Así, la llamada al método **`sacarDeDeposito(100);`** podré incluirla en una expresión lógica, que me permita actuar en consecuencia.

```
if (sacarDeDeposito(100))
     System.out.println ("Hemos podido sacar los 100 litros");
else
     System.out.println ("Cantidad insuficiente. No hemos sacado nada del depósito.");
```

Si como resultado de un método

- queremos que éste devuelva un objeto de una clase,
- podemos indicarlo en la definición del método escribiendo el nombre de la clase delante del nombre del método,
- como si se tratara de cualquier otro tipo.
- Pero lo que se pasa como valor de retorno no es el valor de la variable, sino la referencia al objeto, es decir la dirección de memoria en la que se encuentra el objeto.
- Debemos tener en el lugar que se hizo la llamada alguna referencia del mismo tipo que recoja ese valor, apuntando al objeto modificado que el método devuelve, para que no se pierda ese valor.

Por ejemplo.

Suponemos que un **nombre completo** consta de

- una o dos palabras que son el **nombre**, separadas por blancos, y seguidas de otras dos palabras que son los apellidos, también separadas por blancos.
- Los **apellidos** serán todos los caracteres que hay desde el segundo blanco, contado desde el final del nombre hasta el final del nombre.
- No se tienen en cuenta los posibles espacios en blanco finales, ni se contempla la posibilidad de nombres con un solo apellido (habría que meterle un guión como segundo apellido si es extranjero para indicar que no tiene, o algo así).
- También deberíamos tener en cuenta que no puede haber más de un espacio en blanco separando los dos apellidos.

```
public String apellidosDe(String nombreCompleto){
        String apellidos;
   /* Quitamos los blancos iniciales y finales que tenga el nombre.*/
        nombreCompleto=nombreCompleto.trim();
   /* Inicializamos posicion a la del último carácter del nombre */
```

```
        int posicion= nombreCompleto.length()-1;
        /* Inicializamos los blancos encontrados a cero */
             int nBlancos=0;
        /* Recorremos el nombre carácter a carácter, desde el último hacia atrás mientras no
         * hayamos llegado al primer carácter ni hayamos encontrado el segundo blanco contando
         * desde el final, que es el que separa el nombre de los apellidos.
         */
             while(posicion>0 && nBlancos!=2){
           /* Si el carácter de la posición es un blanco...*/
                     if(nombreCompleto.charAt(posicion)== ' '){
                /* ... sumamos uno al número de blancos encontrados*/
                               nBlancos++;
                     }
           /* En cualquier caso seguimos comprobando la posición anterior */
                     posicion--;
             }
        /* Obtenemos la subcadena que comienza en la posición siguiente al segundo blanco
         * contado desde el final, que es la primera letra del primer apellido y que llega hasta
         * el final del nombre.
         */
             apellidos = nombreCompleto.substring(posicion+1);

        /* Devolvemos el objeto String que hemos identificado como apellidos a partir del nombre
         * recibido como parámetro.
         */
             return apellidos;
    }
```

Una prueba de su uso:

```
 ...
String nombreTrabajador="José Luis López Martínez    ";
String apellidosTrabajador= apellidosDe(nombreTrabajador);
System.out.println("Los apellidos de "+nombreTrabajador+" son: "+apellidosTrabajador);
...
```

Por último mencionar la excepción aparente: **los constructores.**

Los métodos constructores son los únicos para los que no hay que especificar ningún tipo devuelto, ni siquiera `void`.

¿Y eso por qué?

Recuerda que los constructores se identifican por llamarse exactamente igual que la clase, y que se usan para crear objetos de esa clase al ser invocados mediante el operador **`new`**. **No es que los constructores no devuelvan nada. Es que ya se sabe implícitamente que devuelven un objeto, justamente el que están creando, y que ese objeto va a ser de la clase que tiene el mismo nombre que el constructor.**

4.2. Paso de parámetros a métodos

Paso de parámetros a métodos

Recuerda que los parámetros son los que permiten que el método no se ejecute exactamente igual cada vez. Es la forma de "particularizar" cada ejecución. De momento podemos decir que 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 (cuando se trate de este caso).

¿Cómo se pasan los parámetros a los métodos? ¿Hay más de una forma de hacerlo?

En muchos lenguajes 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)**, de forma que todas las operaciones o modificaciones que se hacen sobre el parámetro dentro del método se pierden cuando termina la ejecución del método, no se ven desde fuera. **Es la forma en que se pasan los parámetros en Java para tipos de datos primitivos.**

  DEMO: Visualiza el flujo de ejecución y el estado de la memoria en un paso de parámetros por valor

- **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 y cualquier modificación se hace sobre el objeto original. Cuando el método termina, se borra la copia de la referencia que se pasó como parámetro, pero la referencia original sigue apuntando al objeto sobre el que ha trabajado el método. **Ésta es la forma en que Java usa como parámetros todos los objetos o instancias de clase**. (realmente puede considerarse que en el paso de un parámetro por referencia, lo que estamos haciendo realmente es pasar por valor la dirección de memoria a la que apunta la referencia)

  DEMO:Visualiza el flujo de ejecución y el estado de la memoria en un paso de parámetros por referencia

Un **error** muy común en el paso de parámetros (tanto por referencia como por valor), se suele producir:

- al declarar variables dentro del método con el mismo nombre que los parámetros recibidos,
- lo que conlleva una nueva reserva de memoria y un tratamiento diferente para las dos variables con el mismo nombre.

Veamos un ejemplo de este error que aparece con cierta frecuencia entre los programadores noveles.

DEMO: Visualiza un ejemplo de paso de parámetros por referencia erróneo

Tanto el tipo devuelto, como el nombre y la lista de parámetros que se le pasan entre paréntesis, conforman la signatura de un método. Es lo que nos permite distinguir a unos métodos de otros, y constituye su interfaz, la forma de comunicarnos con él como usuarios programadores.

En los siguientes apartados, una vez que hayamos visto las diferencias entre tipos básicos y tipos por referencia, profundizaremos en este tema mediante ejemplos.

Autoevaluación

5. Cadenas de caracteres

Cadenas de caracteres

*Ahora que **Víctor** ya sabe cómo está formado un programa en Java, **Carmen** decide proponerle la construcción de un **ejemplo** sencillo mediante el que pueda practicar la definición y llamada a métodos.*
*Pero opina que sería ideal que al mismo tiempo que emplea variables de **tipos primitivos**, utilice también **cadenas de caracteres**, ya que suele ser muy habitual utilizar variables para almacenar nombres y valores con los que no se van a realizar operaciones.*
*Por ello decide iniciarle en las **clases String y StringBuffer,** casi imprescindibles para cualquier programador en Java y que en adelante las va a utilizar casi en todos los métodos que programe. Además considera muy importante comentarle el concepto de tipos por referencia como mecanismo de gestión de memoria ideal en Java.*

Recuerda de las unidades anteriores:

- que las cadenas de caracteres nos permitían representar textos,
- sucesiones de caracteres del alfabeto,
- como pueden ser nombres, frases, mensajes, o el texto de un capítulo del Quijote.

Cualquier aplicación usará en numerosas ocasiones **cadenas de caracteres**, en los mensajes que envía al

usuario para interactuar con él, en las direcciones de los empleados de una empresa, en el nombre y los apellidos de esos usuarios, en la descripción del puesto de trabajo, etc.

Son de hecho tan frecuentes, que 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`**.

Sus características, sus similitudes y sus diferencias son el grueso de los siguientes apartados. 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.

No obstante, como todo lo que vamos a decir para **`StringBuffer`** es exactamente igual para **`StringBuilder`**, y como **`StringBuffer`** está presente en todas las versiones anteriores del JDK, vamos a referirnos a ella. Si tienes una versión reciente del JDK, podrás cambiar tranquilamente todas las apariciones de **`StringBuffer`** por **`StringBuilder`**, ya que ambas clases tienen exactamente los mismos métodos.

La clase **`String`** es la primera que usaremos para definir un tipo "por referencia" y vamos a usarla para explicar este concepto.

5.1. Tipos básicos frente a tipos por referencia

Tipos básicos frente a tipos por referencia

¿Cuál dirías que es la característica común a todos los tipos primitivos de datos?

Que todos los tipos primitivos tienen un tamaño fijo, conocido de antemano. Por ejemplo,

- sabemos que el tipo **`int`** ocupa 32 bits,
- y que el tipo **`double`** ocupa 64 bits, etc.
- Siguiendo con el tipo **`int`**, sea cual sea el valor **`int`** que tome una variable, sabemos que va a ocupar exactamente 32 bits.

Esta característica es fundamental. Cuando declaramos una variable de un tipo básico, buscamos memoria libre donde quepa cualquier dato de ese tipo. En el caso de **`int`**, buscaríamos una zona de memoria de 32 bits libres, y esa zona la asociamos permanentemente con el nombre de la variable.

Si queremos **cambiar** el valor de la variable basta con acceder a esa posición y escribir el nuevo valor. Sea cual sea, seguro que ocupa exactamente el espacio disponible para la variable.

- **Esto permite que el compilador asocie el identificador de la variable**
- **con la dirección de memoria donde se almacena su valor**,
- **pudiendo sustituir al compilar todas las alusiones a ese identificador por su dirección de memoria**, que siempre va a ser la misma, porque en ella cabe cualquier valor posible.

DEMO: Visualiza un ejemplo de gestión para guardar variables siguiendo el modelo para tipos básicos

¿Qué ocurre con los tipos por referencia?

En general, que no podemos saber el tamaño exacto que va a ocupar cada objeto de una clase (tipos por referencia). Por ejemplo, para los objetos de tipo **`String`**, que son cadenas de caracteres, podemos tener valores muy diferentes. Si cada carácter Unicode ocupa 16 bits, mientras más larga sea una frase, más espacio de memoria necesitará para almacenarse.

Por ejemplo:

```
String texto= new String("Pepe");
```

La variable texto ocupa 4x16 =64 bits en memoria. Pero podemos cambiar el valor de esa variable:

```
texto= "Nicomedes Gumersindo Iraolagoitia Zumalacandarría"
```

Ahora texto ocupa 49x16=784 bits en memoria (el espacio en blanco también es un carácter que ocupa sus correspondientes 16 bits).

Pero la variable de tipo String texto puede contener un valor mucho más largo:

```
 texto= "Este es un ejemplo de que el espacio
que ocupa un String puede variar considerablemente, sin que podamos
establecer un tamaño estándar"
```

Ocupa bastante más espacio. Y seguro que podemos pensar en textos mucho más largos, ¿verdad?

5.1.1. Espacio que debemos reservar

Espacio que debemos reservar

Puestas así las cosas, ¿qué espacio debemos reservar para almacenar cada literal o cada objeto de tipo **`String`**? En principio se podría pensar en dos alternativas:

- **Reservar para cualquier variable de tipo `String` un mismo tamaño fijo**, lo suficientemente grande como para que no tengamos problemas al representar cualquier valor. **Inconveniente: en la mayoría de los casos estaríamos desperdiciando mucha memoria, y en algunos pocos casos, ni siquiera sería suficiente la memoria reservada**. En algunos lenguajes, ésta es la opción elegida.
- **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`**). Cuando se declara una variable por referencia, en ella no se almacena el valor que nos interesa guardar, sino la dirección de memoria donde habrá que buscar el valor, donde realmente se almacena el valor. **Con esto conseguimos que al compilar la variable se ligue con una dirección fija,** al igual que con los tipo primitivos**, pero en esa posición no buscamos el valor que nos interesa, sino la dirección en la que va a estar ese valor (la referencia). Además ese valor estará ocupando exactamente el espacio que necesita.**

¿Qué ventaja tiene el uso de referencias? Veamos como funcionan, y lo entenderás:

- Cuando ejecutamos la primera sentencia en la que **`texto`** toma el valor **`"Pepe"`**, **al declarar esa variable**, se busca una zona de memoria en la que quepa justamente eso, una dirección de memoria (**las direcciones de memoria siempre ocupan el mismo tamaño**, a fin de cuentas son números). Supongamos que la encuentra en la dirección de memoria 3000.
- Etiquetamos la dirección encontrada con el nombre **`texto`**, y eso ya no va a cambiar. Siempre que en el programa aparezca el identificador **`texto`**, el compilador lo va a sustituir por la dirección 3000, para que el programa acceda a esa posición en busca del valor de **`texto`**.
- Cuando el constructor **`new`** crea el nuevo objeto **`String`**, **busca espacio libre en memoria donde poder alojar cuatro caracteres, los necesarios para el literal `String "Pepe"`**. Supongamos que lo encuentra en la dirección de memoria 5000.
- En esa posición crearía el objeto, de forma que la zona de memoria que comienza en la posición 5000 contendrá el literal **`"Pepe"`**.
- Pero necesitamos ligar el identificador **`texto`** con el valor **`"Pepe"`**, por lo que **en la dirección 3000 lo que guardamos realmente es el valor 5000**. El programa al ejecutarse accederá a la posición 3000 y al encontrar allí almacenada la dirección 5000, accederá a ésta en busca del valor para **`texto`**.
- Cuando posteriormente cambiamos el valor de **`texto`** por otro más largo que ocupa 784 bits, (**`"Nicomedes Gumersindo Iraolagoitia Zumalacandarría"`**) **buscamos un nuevo trozo de memoria libre donde quepa el nuevo valor**. Supongamos que lo encuentra en la dirección de memoria 6000.
- Guarda empezando en la posición 6000 el nuevo valor, **`"Nicomedes Gumersindo Iraolagoitia Zumalacandarría"`**, que ocupa los 784 bits que necesita.
- Actualiza la referencia, es decir, guarda el valor 6000 en la posición 3000. De esta manera, cuando encontremos en el programa el identificador **`texto`**, iremos a su dirección asociada, que es la 3000, y veremos que contiene el valor 6000, que es la dirección a la que iremos a buscar el valor actual para **`texto`**.

- Cualquier nueva actualización que modifique el tamaño del objeto, alojará ese nuevo objeto modificado en una nueva zona de memoria libre y actualizará la referencia, pero nos da lo mismo la dirección concreta que sea. El programa siempre irá a consultar la dirección en la que está el objeto a la dirección 3000.
- **Por tanto, no es necesario limitar el tamaño. Cuando cambia el objeto, se busca un nuevo espacio en memoria donde poder alojarlo de forma que ocupe sólo el espacio que necesita, y actualizamos el valor en la referencia.**

DEMO: Visualiza un ejemplo de gestión para guardar variables siguiendo el modelo por referencia.

5.2. El recolector automático de basura (garbage collector)

El recolector automático de basura (garbage collector)

¿Te has preguntado qué pasa con el valor anterior, es decir, con el objeto antiguo, una vez que actualizamos la referencia?

Realmente no lo hemos borrado de la memoria. Siguiendo con el ejemplo del apartado anterior, cuando en la posición 3000 cambiamos el valor de 5000 a 6000, no hemos cambiado nada en la posición 5000, que sigue conteniendo el mismo valor que tenía. ¡Si no nos ocupamos de esto, la memoria se va a llenar con valores viejos, que ya no utilizamos!

Efectivamente, habrá que liberar el espacio de memoria que ocupaban esos viejos objetos, que ya no son útiles, dejándolo disponible para cualquier otro propósito.

En algunos lenguajes, como C, es el programador el que debe tener cuidado con destruir esos objetos, liberando la memoria que ocupaban. Y si no haces las cosas bien en esos lenguajes, puedes haber dejado un objeto desconectado, sin ninguna referencia, lo que haría que ya tampoco pudiéramos llegar a él para borrarlo.

Si un método encargado de borrar objetos tiene este fallo, se convierte en una especie de agujero negro que va absorbiendo cada vez más memoria, hasta que no queda memoria libre para que el programa siga funcionando, y aborta, o se bloquea el ordenador.

Afortunadamente para ti, las cosas en Java son bastante más fáciles.

Justamente, lo único que hacemos es actualizar la referencia, de forma que no queda ninguna referencia que apunte al objeto antiguo. Eso consigue eliminar el objeto antiguo:

- **Java dispone de un programa de baja prioridad que se ejecuta en segundo plano, y que mantiene una lista actualizada de todas las referencias.**
- **Cuando encuentra una dirección de memoria ocupada a la que no apunta ninguna referencia, la identifica como basura, y la libera.**
- **Si no hay ninguna referencia, no hay forma de acceder a ese objeto, por lo que a todos los efectos, es imposible volver a usarlo. Así que realmente es adecuado considerarlo como basura y eliminarlo automáticamente.**
- Ese programa es **el recolector automático de basura** (**garbage collector**).
- **El recolector de basura se activa cuando la máquina virtual Java lo estima oportuno**, sin que nosotros intervengamos para nada.
- De hecho, **aunque existe un método `gc()` para activar el recolector de basura** cuando detectamos que se está produciendo mucha basura en nuestro programa, **no es más que una recomendación a tener en cuenta por la máquina virtual, que no garantiza para nada que el recolector de basura se active precisamente en ese momento**.

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. Además, la comprobación de las referencias para ver si una zona ocupada de memoria es o no basura,

también consume un cierto tiempo de procesamiento, incidiendo en la eficiencia. Nada es gratis, pero en este caso el precio merece la pena. Además, esa pérdida de eficiencia es perfectamente asumible si tenemos en cuenta la potencia de procesamiento de los ordenadores actuales.

Autoevaluación

5.3. La clase String

La clase String

Ya hemos visto que las variables de tipo **`String`** son variables por referencia, y que nos permiten representar cadenas de caracteres. Pero hay alguna pequeña diferencia con respecto a otras clases. Cualquier otra clase 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.

Por eso tengo **dos formas** de crear objetos de tipo **`String`**:

- Como si de un tipo primitivo se tratase. 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");`

Las dos sentencias anteriores son, a todos los efectos, equivalentes.

Existen muchos constructores más de la clase **`String`**, pero por ahora no los usaremos, ya que la mayor parte de ellos, construyen un **`String`** a partir de un array de **`char`**, o a partir de un array de **`byte`**. Aún no nos hemos familiarizado con los arrays en Java, por lo que difícilmente podemos usar aún esos constructores.

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.

¿Qué quiere decir esto? Aparentemente podemos modificarlos. Si yo escribo:

```
String a= "Hola"
a= a+ " desde Java";       //El operador + para String concatena las dos cadenas.
System.out.println(a);
```

El resultado es un mensaje escrito que dice:

```
Hola desde Java.
```

Aparentemente sí que se ha modificado el valor de la variable **`a`**, no parece inmutable. Pero sólo es aparentemente. 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**.

Sí, con la misma referencia, **`a`** llego al nuevo objeto, y por eso parece que lo he modificado, pero lo que he hecho es crear otro nuevo. Mientras, el viejo objeto será tarde o temprano eliminado de la memoria por el recolector de basura. Volveremos sobre esto al hablar de las diferencias con **`StringBuffer`**.

5.4. La clase StringBuffer frente a la clase String

La clase StringBuffer frente a la clase String

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);

//Las sentencias anteriores son equivalentes a la siguiente, que ponemos comentada:
//StringBuffer textoDocumento=new StringBuffer("Vamos a escribir un texto para procesarlo:");
```

El resultado es que se crea un **nuevo** objeto (una nueva cadena de caracteres) de tipo **StringBuffer** que tiene justamente ese texto.

¿Es que tiene algo malo la clase **String**, que hace necesario el uso de otra clase para las cadenas de caracteres?

Bueno, no es algo malo ni bueno, sino algo diferente, que hace que la clase **String** no siempre sea la más adecuada para trabajar con cadenas de caracteres. Esa característica es justamente la de **inmutabilidad** que hemos comentado en el apartado anterior.

Recuerda que las cadenas de caracteres se usan mucho, tanto como para que incluso la clase **String** tenga su propia forma abreviada de invocar a los constructores. Por tanto es muy frecuente trabajar con cadenas de caracteres. Cada vez que modificamos una variable de tipo **String** tenemos que buscar un nuevo espacio de memoria libre para el nuevo objeto modificado y actualizar la referencia. Esto conlleva un cierto tiempo de procesamiento. **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.**

Por eso en Java se proporciona otra clase para crear y manipular cadenas de caracteres: La clase **StringBuffer** (o su equivalente **StringBuilder**)

StringBuffer define objetos que sí pueden ser directamente modificados, sin necesidad de crear un nuevo objeto, ni de actualizar las referencias.

- Eso es posible porque la clase **StringBuffer** reserva para cada cadena de caracteres un espacio extra de almacenamiento, además del estrictamente necesario.
- Es espacio extra que se puede usar para hacer modificaciones que alteren el tamaño de la cadena de caracteres, siempre y cuando no impliquen necesidades de espacio mayores del espacio extra adicional que reserva. Si fuera así, seguiría siendo necesario realojar el objeto en una nueva zona de memoria, en la que quepan todos los cambios, y actualizar la referencia.

El espacio extra adicional que reservamos depende del tamaño de la cadena de caracteres que introduzcamos. Si esa cadena crece mucho más allá del espacio extra almacenado, el nuevo espacio extra que se reservará será también mayor.

¿Qué conseguimos con esto?

- 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. Pero nada es gratis en programación. A cambio estamos admitiendo un uso menos eficiente de memoria, al reservar para cada cadena de caracteres concreta más espacio del estrictamente necesario.

Un **ejemplo** puede ser el texto de un documento en un procesador de textos. Cada vez que escribimos o borramos una letra, o que modificamos otra, estaríamos modificando el contenido de la variable de tipo cadena de caracteres que almacena en memoria ese texto. Y con un procesador de textos, estas operaciones son constantes, por lo que interesa maximizar su tiempo de respuesta, aunque sea a costa de dedicar un poco más de memoria de la necesaria. **Elegiremos en este caso declarar la variable como de tipo StringBuffer**.

- Si para la **variable de tipo cadena de caracteres** en cuestión sabemos que **rara vez va a ser necesario hacer alguna modificación,** lo lógico será declararla como de **tipo String**. Estaremos **optimizando** el uso de la memoria, y tampoco estamos perdiendo eficiencia en el tiempo de

procesamiento de los cambios, por la sencilla razón de que rara vez habrá cambios.

Un ejemplo puede ser la variable que usamos en un programa para almacenar el nombre de una persona. Las personas no suelen cambiar de nombre con frecuencia. De hecho, no suelen cambiar de nombre en toda su vida, una vez que se les asigna uno al nacer. Una modificación sobre el nombre de la persona parece poco probable. A lo sumo, corregirlo alguna vez, si es que nos equivocamos al introducirlo, o pasarlo a mayúsculas para guardarlo por primera vez en un fichero o en la base de datos, para que todos los nombres estén siempre almacenados en mayúsculas. (esto facilita las búsquedas, ya que "josé" y "JOSÉ" son en principio para el programa nombres distintos). **Elegiremos en este caso declarar la variable como de tipo `String`**.

A continuación te proponemos un ejemplo en el que se aprecia que los objetos de la clase `String` no se modifican, sino que se sustituyen por otros, mientras que los objetos `StringBuffer` sí son directamente modificados. Ejecútalo con NetBeans, y lee atentamente los comentarios que incluye el código. Te ayudarán a entender mejor lo que se quiere ejemplificar.

Descarga el archivo StringFrenteStringBuffer.java

5.5. Métodos disponibles, y ejemplos de uso

Métodos disponibles, y ejemplos de uso

¿Qué **operaciones** podemos hacer con cadenas de caracteres?

Muchas, y muy variadas. A lo largo de tu vida como programador, verás que cualquier programa que hagas suele necesitar usar y manipular cadenas de caracteres.

La clase `String` viene con un considerable repertorio de métodos disponibles, que permiten hacer prácticamente cualquier cosa a la hora de manipular una cadena de caracteres. No pretendemos verlos aquí de forma exhaustiva. No es necesario, ni tiene sentido.

Lo que sí tiene sentido es que si te piden hacer alguna operación con cadenas de caracteres, previamente **consultes la API de Java** (Application Programer's Interface), y compruebes si esa operación ya está disponible en la clase `String`, proporcionada por el lenguaje, o si puedes construirla fácilmente a partir de los métodos disponibles.

Esto es así para la clase `String`, y para cualquier otra clase de las muchas que nos proporciona el lenguaje. Casi podemos decir que cuando nos planteemos hacer algo, lo primero que tendríamos que hacer es preguntarnos si no habrá ya una clase en la librería de Java (JFC o Java Foundation Classes) que haga esa tarea. Un vistazo previo a la documentación de las clases (API) puede ahorrarnos tiempo y quebraderos de cabeza.

Por eso vamos sencillamente a nombrar algunos de los métodos más necesarios para manipular cadenas de caracteres, tanto de la clase `String` como de la clase `StringBuffer`, y vamos a usar algunos de ellos en nuestros ejemplos.

PARA SABER MÁS:
Para una lista detallada de los métodos disponibles y de su utilidad, es obligatorio consultar la API, que puedes consultar directamente en la siguiente página Web:
Documentación de la API de Java

ZONA DE DESCARGA
También puedes descargarla e instalarla en tu disco duro, para poder consultarla sin conectarte a Internet, desde el siguiente enlace. El proceso de descarga es similar al del JDK, descargando un fichero ejecutable, que al ejecutarse instala la documentación.
Instalación de la Documentación de la API de Java
Resaltamos en rojo en esta imagen la parte de la pantalla en la que encontrarás el vínculo para descargarte la documentación.
Ten en cuenta que estos enlaces pueden cambiar ligeramente si aparecen nuevas versiones del JDK, pero que en cualquier caso siempre los encontrarás en la página oficial de Sun MicroSystem.

5.5.1. Métodos de uso más frecuente para la clase String

Métodos de uso más frecuente para la clase String

La mejor manera de presentarte estos métodos puede ser en una especie de tabla, tal y como aparecen en la API, pero no te vamos a copiar esa información. Sólo **pretendemos que dispongas de la lista a mano, y en español, para aquellos métodos que pueden usarse con más frecuencia**.

Para guardar la similitud, hemos respetado en inglés los nombres de los argumentos de la llamada a los métodos.

Pulsando sobre el nombre del método, irás directamente a la página Web que ofrece información detallada de cada uno.

Accede a la documentación

Autoevaluación

5.5.2. Métodos de uso más frecuente para la clase StringBuffer

Métodos de uso más frecuente para la clase StringBuffer

Como es natural, la clase `StringBuffer` presenta algunos métodos bastante parecidos a los de la clase `String.` A fin de cuentas, ambas se usan para representar y manipular cadenas de caracteres. Es por eso que para aquellos que son similares no se va a volver a poner la descripción completa de su función, ni ejemplos, ya que prácticamente la única diferencia sería la declaración de las variables, que serían de tipo `StringBuffer`.

En cualquier caso, **la principal novedad que aporta la clase StringBuffer frente a String, además de crear objetos inmutables, es la incorporación de los métodos `append()` e `insert()`, que están sobrecargados para aceptar datos de cualquier tipo**. Decir que están sobrecargados es decir que tenemos más de un método que se llama igual, pero que se diferencian por la lista de parámetros que admiten. **Para el programador, todos esos métodos se usan como uno sólo al que es posible pasarle parámetros de distinto tipo**. En la tabla que aparece más adelante, verás el significado del concepto de sobrecarga con el método `append()`y con el método `insert()`.

Al igual que en el caso de la clase `String`, sólo pondremos en la tabla, a modo de referencia rápida, los que por ahora merece la pena destacar. No obstante, en este caso, nos hemos dejado muchos métodos fuera de la tabla. Para una descripción completa, debes acudir a la API de Java, disponible en la Web.

Allí podrás leer también que:

- **a partir de la versión JDK 5,**
- **la clase `StringBuffer` puede ser sustituida, y de hecho se aconseja hacerlo en la mayor parte de los casos**
- **por la clase `StringBuilder`,**
- **que soporta todas las mismas operaciones,**
- **pero que es más rápida** al no garantizar sincronización entre varios `Threads`.

Como la funcionalidad de `StringBuilder` es exactamente la misma que la de `StringBuffer`, y conceptualmente no tienen apenas diferencias, reproducimos aquí la tabla de métodos para `StringBuffer`, que sí existe para todas las versiones del JDK. La de `StringBuilder` sería idéntica, pero cambiando `StringBuffer` por `StringBuilder` allí donde aparezca.

Pulsando sobre el nombre del método, irás directamente a la página Web que ofrece información detallada de cada uno.

Accede a la documentación

Autoevaluación

5.5.3. Ejemplos de uso con String y StringBuffer

Ejemplos de uso con String y StringBuffer

En este apartado vamos a presentarte algunos **ejemplos de código** en el que se utilicen las clases **`String`** y **`StringBuffer`**, con el propósito de que consolides la comprensión de las operaciones con cadenas de caracteres, al mismo tiempo que te sigues ejercitando en el uso de estructuras de control del flujo del programa.

También aprovecharemos para comentarte sobre el código algunas características que aparecen en él, como es el orden de evaluación de las condiciones, que no es indiferente, o el uso de caracteres (**`char`**) como expresión a evaluar en las sentencias **`switch`**.

El **primer ejemplo** pretende familiarizarte con el uso y la manipulación de cadenas de caracteres. Consiste en indicar, dada una cadena introducida desde teclado, si es un palíndromo o no. Un **palíndromo** es una frase que al invertirla carácter a carácter, sin tener en cuenta los espacios en blanco ni los acentos, no cambia. Quizás uno de los más famosos palíndromos es el siguiente:

"DÁBALE ARROZ A LA ZORRA EL ABAD"

Para comprobar si una frase es un palíndromo es necesario:

- quitarle primero los espacios en blanco
- e invertirla después,
- para compararla con el original.

Si **coinciden**, será un palíndromo, si no coinciden, no lo será.

Ten en cuenta que el ordenador considera caracteres distintos las letras acentuadas, por lo que en nuestro ejemplo, introduciremos siempre los caracteres sin acentuar, para que funcione correctamente. El ejemplo se simplificaría mucho usando el método **`reverse()`** de la clase **`StringBuffer`**, pero vamos a construir nosotros un método parecido llamado **`invertir()`**, para mostrar las operaciones posibles para manipular cadenas de caracteres.

Descarga el archivo ComprobarPalindromo.java

El **segundo ejemplo** consiste en:

- leer una frase desde teclado,
- e indicar el número de caracteres que contiene,
- y decir cuántos son vocales,
- cuantos consonantes,
- cuantos dígitos,
- cuantos espacios en blanco
- y cuantos otros tipos de caracteres.

También debe indicar cuál es el carácter que **más** se repite, y el que **menos** de los que forman la frase. Lee atentamente los comentarios introducidos en el código, para entender el uso que se hace de los distintos métodos de la clase **String** y de la clase **`StringBuffer`** .

ContarCaracteres.java

El **tercer ejemplo** pretende seguir profundizando en el conocimiento del lenguaje y de las distintas funciones disponibles. Consiste en **contar y escribir por separado las palabras que forman parte de una frase introducida desde teclado**. Este método también se simplificaría mucho usando la clase **`StringTokenizer`**, que dispone de métodos para obtener directamente los tokens que forman parte de un **`String`**, o decir cuántos son. Puedes consultar esta clase y sus métodos, así como un ejemplo en la API de Java. Pero aquí vamos a ser más artesanales, con el propósito de que veas las operaciones que podemos realizar con cadenas de caracteres.

ContarPalabras.java

PARA SABER MÁS:
En este enlace encontrarás un resumen de los métodos de las clases String y StringBuffer, así como ejemplos de los mismos.
Clases String y StringBuffer **[Versión en caché]**
En estos enlaces encontrarás ejemplos desarrollados sobre StringBuffer
StringBuffer **[Versión en caché]**
Modificar un StringBuffer **[Versión en caché]**

6. Tablas, matrices, vectores o arrays

Tablas, matrices, vectores o arrays

Víctor *empieza a sentirse cómodo con la elaboración de los ejemplos que su compañera le ha planteado sobre el manejo de cadenas de caracteres.*
Ahora el proceso de codificación de un programa en Java, su compilación y ejecución en el entorno de desarrollo NetBeans, se ha convertido en una tarea más o menos mecánica que le resulta bastante sencilla.
Incluso ha tenido la oportunidad de hacer rectificaciones en su código tras los errores devueltos por el compilador, hasta conseguir que su programa haga lo que tiene que hacer sin errores, ni resultados incoherentes.
Carmen *le explica que ésta es la base de todo programador, familiarizarse con un entorno de desarrollo y utilizarlo de forma adecuada en cada aplicación.*

En la unidad 2, dedicada a tipos de datos, ya explicábamos lo que eran los **arrays**, y mencionábamos un posible uso. En definitiva,

- se trata de una estructura de datos
- que podemos imaginar como un casillero
- en el que cada casilla tiene un número asignado,
- al que llamamos índice, que nos permite acceder a él.

Todos los casilleros pueden guardar un dato individual, y el tipo es el mismo para todos ellos. En esa unidad también veíamos que los **arrays** son un conjunto de datos del mismo tipo que se almacenan en posiciones consecutivas de memoria.

- El nombre o identificador del array se asocia con la dirección de memoria donde comienza de la estructura.
- El índice representa el desplazamiento que hay que aplicar sobre esa dirección para llegar a la dirección que ocupa el elemento que tiene ese índice

Afortunadamente, nosotros podemos manejar arrays sin preocuparnos por las posiciones de memoria donde se almacena cada uno de sus elementos. Todo lo gestiona de forma automática el compilador.

Resumiendo:

- **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.**

¿Y qué **utilidad** tienen los arrays? ¿No es lo mismo crear variables distintas, cada una con su nombre, tantas como casilleros tenga el array?

Pues va a ser que no. Imagina que en un **estudio** estadístico hay que calcular la media, la desviación típica, la correlación, y un montón de valores estadísticos más, calculados sobre un conjunto de datos formado por la edad y la altura de los trabajadores de una empresa de 10 trabajadores.

¿Cuántas variables independientes tendríamos que manejar?

- 2 variables (altura y edad) x 10 trabajadores = 20 variables, por lo que necesitaríamos 20 identificadores distintos y tendríamos que usarlos y manejarlos todos en nuestro programa.
- Bueno, 20 todavía serían manejables, pero ¿que pasaría si la empresa tuviera 100 empleados? ¿Y si los empleados fuesen 1000, o incluso más?
- No hay duda. Hacer un programa que necesitara 2000 variables o incluso más, con sus correspondientes 2000 o incluso más identificadores, sólo para almacenar los datos necesarios sería inmanejable.

¿Cómo lo solucionamos?

- Una posibilidad es crear un array `edadTrabajador` y otro array `alturaTrabajador`, de 1000 posiciones cada uno, y guardar los datos en ellos. Así la edad del primer trabajador se guardaría en la primera posición del array `edadTrabajador`, y la altura en la primera posición el array `alturaTrabajador`. Con sólo esos dos identificadores, combinados con el índice que identifica a cada trabajador, podemos manejar todos los datos.

¿Cómo se llama la edad del trabajador nº 125? `edadTrabajador[125]` ¿Y su altura? `alturaTrabajador[125]`

El nombre de un dato concreto es el nombre de la estructura, con el índice entre corchetes.

En los próximos apartados nos centraremos en la forma que tiene Java de declarar y usar arrays.

6.1. Declaración y dimensionamiento de arrays lineales y multidimensionales

Declaración y dimensionamiento de arrays lineales y multidimensionales

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

Continuando con el ejemplo de los datos estadísticos, podríamos representarlos mediante **dos vectores**,

- uno para almacenar las edades
- y otro para almacenar las alturas.

Así tendríamos dos vectores de una sola fila y tantas columnas como trabajadores necesitáramos "fichar".

No obstante, también sería posible disponer de **un único array bidimensional**, llamado `alturaYEdad`, en el que habría tantas columnas como trabajadores, y dos filas.

- Cada columna representaría los datos de un trabajador distinto,
- mientras que la fila 0 representaría la altura del trabajador de esa columna,
- y la fila 1 la edad del mismo trabajador.

Los arrays bidimensionales son llamados también matrices o tablas.

Pero podemos imaginar **arrays de tres dimensiones**, o incluso más. Conceptualmente estamos acostumbrados a manejarnos con las tres dimensiones que nos da el espacio, y nos cuesta imaginar un ejemplo en el que sea necesario usar más de tres dimensiones, pero para un ordenador o un lenguaje de programación, esto no supone ningún problema. 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.**

Veamos cómo podemos **definir arrays de cualquier dimensión**.

Lo **primero** que tenemos que tener claro es:

- el tipo de los elementos del array que vamos a declarar,
- y cuántos van a ser.

6.1.1. Definición del vector en Java

Definición del vector en Java

¿Cómo definiremos el vector **`edadTrabajador`** para el caso de disponer de 100 empleados?

Parece adecuado almacenar la **edad** de cada trabajador como un **número entero**. Por tanto debemos definir un array de enteros.

```
int[ ] edadTrabajador;
```

La declaración no es más que una referencia que podrá apuntar a arrays de enteros, de cualquier tamaño.

En Java también es posible declarar el array de una **segunda** forma, con los corchetes detrás del identificador del array. Así, la declaración anterior también se podría expresar como:

```
int edadTrabajador[];
```

Esta segunda forma,

- aunque es totalmente equivalente a la anterior,
- se **desaconseja**, ya que es menos clara,
- al parecer que se está haciendo una declaración de una variable de tipo **`int`** con un nombre que termina con corchetes, lo cual no es real.

En la **primera** forma queda más claro que se define una variable de tipo array de enteros (**`int[]`**). Por tanto, te aconsejamos usar siempre la primera forma para la declaración de los arrays.

Con esto hemos declarado la referencia que apuntará al objeto de tipo vector unidimensional de enteros. Pero esa referencia ahora mismo **no apunta** aún a ningún objeto. Todavía no la hemos inicializado, porque no hemos creado ese objeto. Para ello debemos llamar al operador **`new`** de **creación de objetos**, indicándole:

- el tipo de objeto que debe crear,
- cuántos elementos va a tener (la dimensión del vector, que en nuestro ejemplo es 100)
- y haciendo que la referencia apunte a ese objeto.

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

Tras esta sentencia tenemos la referencia apuntando a un nuevo array de 100 elementos de tipo entero, pero cada uno de esos elementos se ha inicializado por defecto al valor cero.

- **En Java, todos los datos numéricos se inicializan a cero,**
- **y todos los datos por referencia se inicializan al valor null**.

El valor null representa al objeto nulo, es decir, es un valor que indica al compilador que la referencia no apunta a ningún objeto, (a ninguna posición de memoria). Evidentemente es posible declarar y dimensionar el array en una sola línea:

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

Es importante indicar que como índices en Java:

- **sólo pueden usarse valores enteros,**
- **y que el primer valor del índice siempre es cero.**
- **Es decir, el primer elemento de cada dimensión estará en la posición cero.**

Así, la edad del primer trabajador almacenada en el array se referencia como:

```
edadTrabajador[0]
```

Y la del último trabajador, suponiendo que son 100 los trabajadores del array, será:

```
edadTrabajador[99]
```

¿Y qué **diferencia** hay con la declaración de un array de más de una dimensión?

No mucha, la verdad. En principio:

- basta con poner más corchetes,
- uno para cada dimensión,
- e indicar el máximo para cada dimensión.

Siguiendo con nuestro **ejemplo**, si queremos almacenar tanto la edad como la altura de los trabajadores en el mismo array, con 100 filas (una para cada trabajador) y dos columnas (la primera para la altura y la segunda para la edad), podemos definirlo de la siguiente forma:

```
int[][] alturaYEdad = new int [100][2];
```

Aunque para el lenguaje es indiferente imaginarlo como de dos filas y 100 columnas o como de 100 filas y dos columnas, normalmente nos referimos a la primera dimensión como el número de filas y a la segunda como el número de columnas.

¿Y si son más dimensiones todavía?

Exactamente igual**. Java no impone ningún límite para el número de dimensiones**.

Por ejemplo, imagina la siguiente situación...En una **empresa** quieren mantener almacenado el número de veces que se ha usado cada máquina de una cadena de montaje para cada día.

- En la cadena hay 3 talleres (primera dimensión).
- Cada taller tiene 5 filas de mesas de trabajo (segunda dimensión),
- y cada fila tiene 4 mesas (tercera dimensión).
- Por último en cada mesa tenemos 2 máquinas (cuarta dimensión).
- Y para cada máquina tenemos que almacenar un número entero, que se incrementará en uno cada vez que se utilice esa máquina.
- Me interesa almacenar la información para las 120 máquinas, pero al mismo tiempo, quiero poder identificar el taller, la fila, la mesa y la máquina concreta a la que se refiere la información.

Una solución a este problema puede plantearse con un array de 4 dimensiones:

```
int[][][][] usosMaquinas = new int[3][5][4][2];
```

De esta manera, con **`usosMaquinas[0][4][2][1] =0;`** estoy inicializando a cero al empezar un nuevo día la segunda máquina (máquina 1) de la tercera mesa (mesa 2) de la quinta fila (fila 4) del primer taller (taller 0). Recuerda que los índices empiezan en cero, de forma que si hay tres talleres, estarán numerados del cero al 2, las cinco filas estarán numeradas del cero al 4, etc.

Y la operación a realizar cada vez que use esa máquina será **`usosMaquinas[0][4][2][1]++;`**

Si por ejemplo hacemos **`usosMaquinas[0][5][2][1]++;`** el programa generará **`ArrayIndexOutOfBoundsException`**, un error que hará que aborte, ya que estoy intentando acceder a la sexta fila (fila 5) del taller 0, y esa fila no existe. **Java, a diferencia de otros lenguajes como C, no permite acceder a una posición de memoria fuera del array,** lo cual es una garantía de que no empezarán a ocurrir cosas extrañas.

6.2. Recorrido y llenado de un array

Recorrido y llenado de un array

Puede que te hayas planteado que si el array es **grande**, como los dos de nuestro ejemplo, que tienen 100 posiciones, puede resultar bastante **engorroso** escribir las 100 sentencias necesarias para introducir el valor de las alturas de cada trabajador en el array alturaTrabajador, y las otras 100 sentencias necesarias para introducir las edades en el array edadTrabajador.

Evidentemente, eso se puede hacer más fácil. El programador no tiene que escribir las 200 sentencias, sino un **ciclo** (bucle) mediante el que

- se van recorriendo "en paralelo" ambos arrays,
- solicitando la altura y la edad de cada trabajador,
- guardando esos valores en la posición correspondiente del vector,
- pasando a la siguiente posición hasta que lleguemos al final del array.

¿Qué ciclos serán los más adecuados para trabajar con arrays? El índice lo usamos para desplazarnos por el array, luego el ciclo se tendrá que repetir una vez para cada valor del índice. Y lo bueno es que sabemos que el índice empieza tomando el valor cero y termina tomando el valor de la dimensión correspondiente. **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.**

En nuestro ejemplo, podríamos recorrer y llenar los dos arrays de edades y alturas de forma independiente, cada uno por separado, leyendo los valores desde teclado con la clase de lectura ES que venimos usando, de la siguiente forma:

```
int[] edadTrabajador = new int[100];
for (int trabajador=0;trabajador<100;trabajador++){
        edadTrabajador[trabajador] = ES.leeNº("Introduce la altura para el trabajador "
                                        trabajador + ": ",0);
}
//...
int[] alturaTrabajador = new int[100];
for (int trabajador=0;trabajador<100;trabajador++){
        alturaTrabajador[trabajador] = ES.leeNº("Introduce la edad para el trabajador "

}
```

En el **método de lectura** de un número entero hemos introducido un cero como segundo parámetro, para evitar que se introduzcan edades o alturas negativas.

Fíjate que la variable de **control** usada en cada for (trabajador) se llama igual, pero no es la misma.

- Se declara dentro de la cabecera del for,
- y por tanto "no existe" fuera del bucle for,
- es decir, se elimina al terminar el for,
- por lo que hay que volver a declararla en el siguiente for.

Naturalmente, podría haberla declarado fuera, pero lo habitual es que la variable de control del bucle, que sólo la usamos para llevar la cuenta de las vueltas que llevamos y las que quedan, no se declare fuera del for, ya que no suele usarse fuera de éste.

Pero también podría llenar ambos vectores en paralelo, usando un único ciclo for para leer en cada vuelta tanto la edad como la altura del trabajador, tal y como aparece a continuación:

```
int[] edadTrabajador = new int[100];
int[] alturaTrabajador = new int[100];
for (int trabajador=0;trabajador<100;trabajador++){
        edadTrabajador[trabajador] = ES.leeNº("Introduce la altura para el trabajador "
                                        trabajador + ": ",0);
        alturaTrabajador[trabajador] = ES.leeNº("Introduce la edad para el trabajador "
                                        trabajador + ": ",0);
}
```

En el caso de haber almacenado los datos de la altura y la edad en el mismo array bidimensional **`alturaYEdad`**, el recorrido para introducirlos se haría mediante **un bucle for** en el que se moviera por todos los trabajadores, leyendo para cada uno de ellos tanto la altura como la edad. A continuación está el programa que lo haría, seguido de unas explicaciones sobre algunas novedades que incorpora:

```
int[][] alturaYEdad = new int [100][2];
for (int trabajador=0;trabajador<alturaYEdad.length;trabajador++){

        //lectura de la altura
        alturaYEdad[0][trabajador] = ES.leeNº("Introduce la altura para el trabajador "
                                        trabajador + ": ",0);
        //lectura de la edad
        alturaYEdad[1][trabajador] = ES.leeNº("Introduce la edad para el trabajador " +
                                        trabajador + ": ",0);
        }
```

La principal novedad es que se incorpora, además de usar referencias a los elementos usando dos índices, ya que es un array bidimensional, es el uso de la variable **`length`**.

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.

Así, para controlar en el ciclo for cuando hemos recorrido el array completo, usamos como condición de terminación del ciclo **`trabajador<alturaYEdad.length`**.En este caso, **`alturaYEdad.length`** devuelve el número de filas del array.

Hacerlo así en vez de poner directamente 100 como en los ejemplos anteriores tiene sus ventajas.

- **Si la referencia al array pasara a apuntar a otro array de distinto número de elementos, todo nuestro programa funcionaría igual, sin que fuese necesario actualizar el código para el nuevo tamaño del array.**
- **Cuando le paso un array como parámetro a un método para que haga algo con él, no tengo que pasarle como parámetro además su tamaño**. Esa información va incluida en el atributo **`length`** del array, y se le puede "preguntar" su valor desde el método.
- **También es posible preguntarle a un array por el número de columnas que tiene, o por el tamaño de cualquiera de sus dimensiones,** ya que en el fondo todo array multidimensional no es más que un array de arrays para Java.

En el siguiente apartado veremos con más detalle el uso del atributo **`length`** de los objetos array, junto a la declaración explícita de un array de varias dimensiones, que es un caso más general.

6.3. Arrays multidimensionales de dimensiones variables

Arrays multidimensionales de dimensiones variables

Hemos mencionado que en Java los arrays pueden contener elementos de cualquier tipo. Podemos tener :

- arrays de **tipos básicos**, como en los ejemplos anteriores,
- o arrays de **objetos** de cualquier clase definida por el lenguaje, (String, por ejemplo)
- o de cualquier otra clase definida por el usuario.
- Así podríamos definir una clase **Persona**, y crear un array de personas. Cuando los elementos son objetos, en el array lo que guardamos no son realmente los objetos, sino las referencias a esos objetos.

DEMO: Visualiza un ejemplo de cómo se gestiona la memoria para almacenar un array de múltiples dimensiones

Esa característica es la que usa Java para construir los arrays de más de una dimensión,

- formándolos como un array unidimensional de arrays unidimensionales,
- que a su vez pueden contener arrays unidimensionales, y así sucesivamente.

Así, el array de cuatro dimensiones **`usosMaquinas`** del apartado anterior, para Java es un vector unidimensional de 3 talleres.

- Cada taller es un array unidimensional de 5 filas.
- Cada fila es un vector unidimensional de 4 mesas y
- cada mesa es un vector unidimensional de 2 máquinas, y

- cada máquina viene representada por el dato que necesita almacenar, que en este caso es un número entero, que representa las veces que se ha usado en el día esa máquina concreta.
- De esta forma tenemos el array de cuatro dimensiones de enteros (**`int`**).

Si cada mesa en vez de contener máquinas sobre las que nos interesa sólo un entero (tipo básico) contuviera la información de las 2 personas que trabajan en ella, este último vector unidimensional sería de 2 personas, lo que significaría que en vez de alojar dos enteros almacenaría dos referencias a dos objetos de tipo Persona, que estarían alojados en algún lugar de la memoria. Suponiendo que la clase **`Persona`** ya está creada por el usuario, y suponiendo que el constructor de la clase **`Persona`** necesita como argumentos el **nombre** y la **edad** de la persona, la nueva declaración del array sería algo como lo que sigue:

```
Persona[][][][] trabajadoresTalleres = new Persona[3][5][4][2];
```

Y a cada posición del array se le asignaría valor con sentencias del tipo:

```
trabajadoresTaller[1][2][3][0] = new Persona("Francisco Fernández",38);
```

Eso significa que en el taller 1 (segundo taller), en la fila 2 (tercera fila), en la mesa 3 (cuarta mesa), el trabajador 0 (primer trabajador) se llama Francisco Fernández, y tiene 38 años.

Pero también podemos crear **arrays** de varias dimensiones que **no sean "rectangulares**", es decir, en los que cada fila pueda tener un número distinto de columnas. En general, cada elemento de una dimensión puede ser un array de dimensión distinta.

Para que sirva como demostración y a modo de ejemplo, vamos a crear un vector de **`String`** con forma de escalera que representa los **nombres de las personas** que trabajan en una empresa, de forma que

- las **filas** representan a cada departamento y contienen los trabajadores de cada departamento,
- y las **columnas** representan el nivel jerárquico dentro del mismo.

Debes leer atentamente los comentarios que se insertan. En ellos se explica la forma de crear estos arrays de dimensiones variables, así como la forma de usar la función length para asegurarnos de que nunca nos salimos de los límites del array.

Descarga el archivo MatrizEscalera.java

También es posible inicializar un array, sea de la dimensión que sea, de forma explícita, indicando cuales son sus elementos. Al hacerlo así, Java nos permite una escritura más abreviada, sin tener que hacer llamadas explícitas al operador new y sin tener que indicar la dimensión del vector, ya que implícitamente queda definida. Al declarar explícitamente el array, estamos dando la lista de elementos que contiene, y por tanto queda claro cuantos son, que es lo que indicaría la dimensión.

¿Cómo se inicializa explícitamente el array?

Indicando 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.

Por ejemplo, el array matrizEscalera del ejemplo anterior se podría haber declarado e inicializado de forma explícita tal y como se expresa en el ejemplo siguiente, en la clase MatrizEscarleraExplicita:

Descarga el código fuente: MatrizEscaleraExplicita.java

Todavía vamos a incluir un ejemplo más en el que se definen arrays de varias dimensiones, para ejemplificar varios aspectos. Uno de ellos es la **inicialización por defecto de los datos de un array**, que se inicializan:

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

Otro aspecto de este ejemplo es el uso de un método para **generar números aleatorios**, que puede resultarnos útil para llenar con valores distintos arrays grandes sin tener que introducirlos todos desde

teclado.

Un tercer aspecto es ver que el tamaño de cada dimensión de un array es algo que no tiene por qué conocer el programador. **El tamaño de un array puede decidirse en tiempo de ejecución, incluso ser aleatorio.**

Por último sirve para ver la **creación de un array de objetos no primitivos, al mismo tiempo que se introduce la clase Integer**, que sirve para **ponerle envoltorio de objeto a los números enteros de tipo int**. **Hay clases similares para todos los tipos básicos**, que añaden funcionalidad que no está disponible para los tipos básicos. Por ejemplo, la clase `Integer` proporciona métodos para pasar un `String` a número entero, y así poder operar con él.

La clase que contiene el código se llama `ArrayMultidimension`, y su código lo puedes consultar en el siguiente fichero:

Descarga el archivo ArrayMultidimension.java

6.4. Procesamiento de un array. Paso de arrays como parámetros

Procesamiento de un array. Paso de arrays como parámetros

Normalmente los arrays los usaremos con la finalidad de almacenar información que necesitaremos procesar con posterioridad. Ese procesamiento puede ser buscar un valor concreto, mostrarla por pantalla, encontrar el menor o el mayor valor, ordenar los valores que contiene o hacer algún cálculo sobre cada uno de ellos, actualizarlos, modificarlos, etc.

En la unidad anterior, ya usábamos vectores para el ejemplo de ordenación rápida, por ejemplo.

En muchos casos, el array que contiene los datos debe pasársele como parámetro a algún método que se encargará de procesarlo, y conviene que veas algún ejemplo. Ya hemos visto en el apartado 3 de esta unidad algunos aspectos relativos a

- las llamadas a métodos,
- al paso de parámetros
- y a los valores devueltos por los métodos.

Pero en el ejemplo que vamos a proponerte, quizás puedas ver cómo funciona todo con mucha más claridad.

En la empresa quieren hacer una **simulación** del proceso de producción de una serie de máquinas en una cadena de montaje. Para ello,

- anotarán cuantos productos fabrica cada una de ellas a lo largo del día,
- simulando la carga de trabajo de cada una mediante la generación de números aleatorios,
- y viendo cómo se comporta el sistema.
- se permite al usuario de la simulación introducir el número de máquinas que se van a usar,
- para ver cómo reacciona el sistema dependiendo del número de máquinas que formen la cadena de montaje.

En un array llamado `produccionMaquinas` se generará aleatoriamente el **número de productos** que ha fabricado cada máquina. Además quieren hacer una aplicación en la que se pueda simular el proceso con varias cadenas de montaje, por lo que necesitarán pasarle como parámetro a un método el array de cada cadena de montaje.

Nuestro trabajo consiste en:

- Hacer el **método** al que se le pasa como parámetro un vector de enteros, y que anota en cada posición un número aleatorio comprendido entre 1 y 25, que será la producción diaria de cada máquina, ya que se considera que 25 es la capacidad máxima de producción de cada máquina, en condiciones ideales, por lo que nunca va a poder producir más de esas unidades por máquina.
- Elaborar el método que imprimirá el contenido de un vector de enteros, para poder ver la producción de cada máquina.
- Encontrar la máquina cuya producción ha sido más baja y que puede estar actuando como cuello de botella en la cadena de montaje.

A continuación te mostramos un **programa** que dispone de esos métodos, y que ejemplifica el paso de arrays como parámetros a métodos, y el procesamiento de un array, en este caso para introducirle los valores y parar encontrar el mínimo de los valores del array. Encontrarás **dos métodos** para llenar el array, ambos correctos, y optar por uno u otro es algo que dependerá de cada caso. También encontrarás comentado un **tercer** método para llenar un array, que sirve de ejemplo de lo que NO se debe hacer, ya que no produciría el resultado deseado. Como siempre, es recomendable que leas los comentarios que se han introducido en el código con atención. Te ayudarán a comprender su funcionamiento, y los conceptos que se pretende explicar.

Descarga el archivo PasoArraysComoParametros.java

Un **ejemplo** de esto puede encontrarlo en esta demo. Te recomendamos que lo revises de nuevo detenidamente para ver el paso de arrays como parámetros.

6.5. Inserción, borrado y búsqueda de elementos en un array ordenado

Inserción, borrado y búsqueda de elementos en un array ordenado

¿Suele ser importante que los elementos de una lista estén ordenados según algún criterio?

Suele ser que sí. Sobre todo cuando manejamos **listas de información** con muchos elementos, el hecho de que estén ordenados facilita bastante las búsquedas de un elemento determinado, ya que en muchos casos no habrá que recorrer toda la lista para saber si el elemento que buscamos está o no está en la lista.

Por eso están ordenadas las palabras de un diccionario, o los nombres del listín telefónico, o los alumnos de una lista de clase, por ejemplo.

Basta con que encontremos una **palabra** que alfabéticamente va después de la que buscamos sin que hayamos encontrado ésta, para que podamos afirmar que esa palabra no aparece en el diccionario, y abandonamos la búsqueda sin tener que leer todo el diccionario. Si las palabras no estuvieran en orden alfabético, tendríamos que revisar todo el diccionario, desde el principio hasta el final, antes de poder asegurar que esa palabra no está en el diccionario.

Lo mismo ocurre en el caso de los arrays. Y además, siempre que queramos:

- borrar,
- consultar o
- modificar los datos de uno de los elementos del array,
- lo primero que tenemos que hacer es buscarlo y localizarlo dentro del array.

Además, si el vector no está lleno, resulta útil que esté "empaquetado", es decir,

- que todos los elementos que contenga estén juntos,
- sin huecos libres, al principio del array,
- y que todos los huecos o posiciones libres del vector estén juntos al final del array.
- De esta forma, cuando buscamos un elemento y encontramos un hueco, también podemos abandonar la búsqueda, ya que a partir de ahí sólo van a quedar huecos.

Pero además, si queremos que el vector esté permanentemente **ordenado** para aprovecharnos de búsquedas más rápidas, que harán que otras operaciones también sean más rápidas, mejorando la eficiencia de nuestra aplicación, debemos preocuparnos de **mantener ese orden** cada vez que insertamos un nuevo elemento y preocuparnos de que el vector quede empaquetado cada vez que borramos un elemento.

En el ejemplo que te ponemos al final de este apartado, se han hecho las operaciones de listado, inserción, borrado y búsqueda dentro de un array de objetos `Persona`.

El array se va a usar para una empresa que quiere inscribir a todos los participantes en **un concurso** de postales Navideñas que organiza para todos los trabajadores y familiares, clientes y proveedores de la empresa.

En el concurso hay distintas categorías por edades, y se quiere que los regalos para los ganadores sean lo

más adecuados posible, por lo que parece oportuno conocer el sexo de los participantes. Con fines estadísticos nos interesa además conocer también cual es la persona de mayor edad y de menor edad que participa en el concurso, así como la edad media de los participantes.

Toda esta información se va a guardar en el **vector personas**. Cada elemento va a ser un objeto de la clase Persona, que la hemos definido también. Cada persona va a tener como campos

- un nombre (String),
- una edad (int)
- y un sexo (char: 'M' mujer, 'H' hombre o 'D' desconocido).

Como nos interesa agrupar a los participantes por categorías, el vector se mantiene ordenado por el valor del campo edad.

En el ejemplo interesa que leas detenidamente los comentarios que explican y ejemplifican los conceptos de la lista que aparece en el párrafo siguiente. Debes tener en cuenta qua algunos de ellos son anticipaciones que se hacen a explicaciones que vendrán en unidades posteriores, como los relativos a la definición de campos privados y las menciones a clases y superclases, o al concepto de sobrecarga, por lo que cabe la posibilidad de que no estén totalmente claros por ahora. No te preocupes en exceso. Al aprender un idioma es bueno oír hablar ese idioma a personas que lo hablan con fluidez, aunque no se entienda todo lo que le dicen a uno la primera vez que los oye. Y un lenguaje de programación, no es más que otro idioma, pero más fácil de aprender.

Los aspectos incluidos en el ejemplo son:

- Declaración de un array de objetos definidos por el usuario, y dimensionamiento en tiempo de ejecución.
- Creación de nuevos objetos con los datos leídos desde teclado.
- Uso del puntero de autoreferencia (`this`), que siempre apunta al objeto actual.
- Recorrido y procesamiento de un array. En el ejemplo, para listar todos sus elementos.
- Inserción de elementos en un array ordenado y empaquetado.
- Borrado de elementos en un array ordenado y empaquetado.
- Búsqueda de un elemento en el array, a través del campo por el que se ordena (edad en el ejemplo).
- Búsqueda de un elemento en el array por un campo que no es el que ordena al vector (por nombre).
- Localización del elemento buscado a través del índice de la posición que ocupa en el vector o a través de una referencia que lo apunte directamente.
- Método `toString()`. Definición y uso. Posibilidad de heredarlo de las superclases, incluso de `Object`.
- Orden conveniente en las subcondiciones que conforman una condición de un bucle.
- Acceso limitado a campos privados a través de métodos públicos: Mejora de la seguridad al impedir errores.
- Sobrecarga de un método (`buscarPersona()`).

Descarga el archivo Persona.java

Descarga el archivo VectorPersonas.java