Ciclo de Desarrollo de Aplicaciones Multiplataforma

Grado Superior

U.T.3. PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

Módulo de Programación

Departamento de Informática

IES Monte Naranco

Los programas realizados mediante POO trabajan con objetos. Todos los objetos pertenecen a una clase.

1. CARACTERISTICAS DE LA POO

- **Abstracción.**

  Nos permite crear modelos genéricos (reutilizables para todos los casos), para ello utilizaremos la definición de clase.

- **Encapsulación**

  Consiste en agrupar atributos y métodos de un objeto en un mismo compartimento.

  Muy relacionado con el encapsulamiento está la ocultación de la información. Consiste en la restricción de acceso al valor de los atributos (declarándolos privados) . No se debe acceder directamente a los datos desde el exterior, se debe hacer siempre a través de los métodos. Así podemos asegurarnos de que un valor no será modificado incorrectamente, por parte de otro programador o usuario. Los modificadores de acceso nos introducen por tanto al concepto de ocultación.

- **Herencia**

  Un objeto puede adquirir las propiedades y comportamientos de otro.

- **Polimorfismo**

  Permite implementar múltiples formas de un mismo método, dependiendo cada una de ellas de la clase sobre la que se realice la implementación.

2. CONCEPTO DE CLASE

Una clase consiste en un plantilla para generar objetos. Se especifican las características comunes de un grupo de objetos por medio de:

- Los **atributos o propiedades:** toda aquella información que me interesa guardar de un objeto. **Ejm:** si tenemos creado un objeto coche sus atributos podrían ser: marca, modelo, matrícula, color, etc.

- Los **métodos:** implementan las acciones que se podrán realizar con un objeto de la clase (comportamiento).

Las clases representan entidades. Pueden ser entes físicos (alumnos) o no (titulación académica).

2.1.- DECLARACIÓN DE UNA CLASE

En la declaración de una clase vamos a encontrar:

Cabecera de la clase. Compuesta por una serie de modificadores (indican el control de acceso), la palabra reservada class, y el nombre de la clase (con la primera letra de cada palabra en mayúscula por convenio).

Los modificadores de clase son: public, final y abstract (final y abstract se verán más adelante). El modificador public indica que la clase es visible, es decir, se pueden crear objetos de esa clase en cualquier otra clase esté o no en el mismo paquete. Si no se especifica ningún modificador (modificador de acceso por omisión), la clase sólo podrá ser utilizada o solo será visible desde clases que estén en el mismo paquete. Sólo puede haber una clase public (clase principal) en un archivo .java. El resto de clases que se definan en ese archivo no serán públicas.

Cuerpo de la clase (encerrado entre llaves { }). En él se especifican los distintos miembros de la clase: **atributos o propiedades** y **métodos**. Es decir, el contenido de la clase.

Una clase puede no contener en su declaración atributos o métodos, pero debe de contener al menos uno de los dos (la clase no puede ser vacía).

La declaración de una clase en Java tiene la siguiente estructura general:

```
[modificadores] class  <NombreClase>// Cabecera de la clase {

    // Cuerpo de la clase
    Declaración de los atributos
    Declaración de los métodos
}
```

Ejm:

```
public class Punto {
   // Atributos
  private  int x,y;
    // Métodos
  public  int obtenerX () {
        return x;
   }
```

```
  public  int obtenerY() {
          return y;
     }
 public  void establecerX (int vx) {
          x= vx;
    }
   public void establecerY (int vy) {
           y= vy;
     }
}
```

Cualquier objeto de la clase punto que sea creado almacenará en su interior dos números enteros (x,y), que podrán coincidir o no con el contenido de otros objetos de esa misma clase.

Para identificar una clase podemos hacernos la siguiente pregunta: ¿podemos pensar en objetos concretos de esa clase?. Si no se puede es una clase sospechosa.

El método main lo situaremos en una clase independiente destinada exclusivamente a contener este método, aunque no es obligatorio.

2.2.- ATRIBUTOS O PROPIEDADES

Los **atributos** son cualidades o características cuyos valores son almacenados por los objetos de una determinada clase cuando son creados. **Ejemplos:** los años de una persona, dni.... El volante de un coche no sería un atributo, no toma un valor concreto.

Un atributo siempre tiene un valor, en caso contrario no estaríamos hablando de atributo.

La sintaxis general para la declaración de un atributo es:

[modificadores] <tipo> <nombreAtributo>;

Los atributos pueden ser de cualquier tipo (primitivos o bien objetos de otra clase).

El nombre suele ir por convenio en minúsculas. En caso de que se trate de un identificador que contenga varias palabras, a partir de la segunda se suele poner la letra en mayúsculas. Por ejemplo: primerValor, valor, puertaIzquierda, cuartoTrasero, etc. Cualquier identificador válido de Java será admitido como nombre de atributo, pero es importante seguir este convenio para facilitar la legibilidad del código. Si los nombres de los atributos son cortos no hay que abreviarlos. Por ejemplo, se debe poner fecha y no fech.

Entre los modificadores de un atributo podemos distinguir:

- **Modificadores de acceso**. Indican la forma de acceso al atributo desde otra clase. Son modificadores excluyentes entre sí. Sólo se puede poner uno.

Los modificadores de acceso disponibles en Java para un atributo son:

- **public**: indica que **cualquier clase** de cualquier paquete tiene acceso a ese atributo. No es muy habitual declarar atributos públicos (public).
- **private:** indica que sólo se puede acceder al atributo desde **dentro de la propia clase**. El atributo estará "oculto" fuera de la clase en la que esté declarado.
- Si no se indica **ningún modificador de acceso** en la declaración del atributo (**por omisión**), se permitirá el acceso a este atributo desde todas las clases que estén dentro del **mismo paquete** (**package**) que la clase que contiene el atributo.
- **protected**: en este caso se permitirá acceder al atributo desde la propia clase, desde cualquier **subclase** (se verá más adelante al estudiar la **herencia**) y desde las clases del **mismo paquete**.

Resumen de los distintos niveles accesibilidad que permite cada modificador:

	Acceso desde la misma clase	**Acceso desde una clase del mismo paquete**	**Acceso desde una clase de otro paquete**
sin modificador	SI	SI	NO
public	SI	SI	SI
private	SI	NO	NO
protected	SI	SI	NO

Lo normal es declarar los atributos como privados, de manera que si se desea acceder o manipular algún atributo se tenga que hacer a través de los métodos proporcionados por la clase que deben estar declarados como públicos, de esta manera evitamos que sean manipulados inadecuadamente desde el exterior del objeto.

• **Modificadores de contenido**. No son excluyentes, pueden aparecer varios a la vez.

Son los siguientes:

- Modificador **final.** Indica que el atributo es una **constante**. Su valor no podrá ser modificado a lo largo de la vida del objeto. No es una constante al uso sino que es inicializada en el momento de crear el objeto, por lo tanto puede ser distinta para cada uno de ellos. Por convenio el nombre de los **atributos constantes** (final) se escribe con **todas las letras en mayúsculas**.

- Modificador **static.** Los atributos que tienen modificador static se denominan atributos de clase, lo que significa que el atributo es común o el mismo para todos los objetos que se creen de una misma clase. Es decir, todas los objetos compartirán ese mismo atributo con el mismo valor.

  Se deberían declarar como static solo aquellos atributos que no tienen demasiado sentido como parte del objeto si no como parte de la clase en sí. Estos atributos existen aunque no exista ningún objeto de la clase y se puede acceder a ellos utilizando tanto métodos estáticos como no estáticos mediante: [**nombre de la clase].atributo.**

  Este tipo de atributos existen siempre que la clase está cargada en memoria (al ejecutar un programa o mediante la sentencia import), aún cuando no exista ningún objeto perteneciente a dicha clase.

Ejemplos de atributo estáticos:

- Un **contador,** que indica el número de objetos de una determinada clase que se han ido creando.

  En la clase Punto del ejemplo, se podría incluir un atributo que fuera un contador para llevar un registro del número de objetos de la clase Punto que se van creando durante la ejecución del programa.

- Supongamos la clase barril de petróleo. Cada barril de petróleo viene identificado por un número y un precio. El número es propio de cada barril (atributo no estático), el precio es propio de la clase barril no del barril concreto que tengo en las manos, sería por tanto un atributo estático. Si sube el precio del barril de petróleo no se sube a un barril sino a todos.

Ejemplo:

```
public class Punto {
// Coordenadas del punto
private int x, y;
private static cantidadPuntos; // Atributo de clase. Cantidad de puntos creados hasta el momento.
```

Cada vez que se cree un objeto de la clase Punto se debe incrementar el valor del atributo cantidadPuntos.

Los atributos o propiedades que no tienen el modificador static se denominan atributos de instancia, lo que significa que únicamente existen en el interior del objeto donde han sido creados y por tanto su vida comenzará cuando el objeto sea creado; así pues, cada vez que se cree un objeto se crearán tantas variables como atributos de instancia estén definidos en la clase.

Fuera del objeto esas variables no tienen sentido, y si el objeto deja de existir esas variables también dejan de hacerlo.

Ejemplos:

```
int x;
public int elementoX, elementoY;
private int x, y, z;
private final float PI =3.1416
```

2.3.- MÉTODOS

Forman parte de la estructura interna del objeto junto con los atributos. Los métodos representan la **interfaz** de una clase. Son la forma que tienen los objetos de comunicarse con otros solicitándole cierta información o pidiéndole que lleve a cabo una determinada acción (cosas que puede hacer un objeto).

Los métodos suelen declararse después de los atributos. Aunque atributos y métodos pueden aparecer mezclados por todo el interior del cuerpo de la clase, es aconsejable no hacerlo para mejorar la **claridad** y la **legibilidad** del código.

La definición de un método se compone de dos partes:

- **Cabecera del método**: contiene un conjunto de posibles modificadores (de acceso o/y de contenido), el nombre del método junto con el tipo devuelto y una lista de parámetros.

La sintaxis general de la cabecera de un método podría quedar así:

[private | protected | public | ninguno] [static | final] <tipo> <nombreMétodo> ([<lista_parametros>])

Normalmente los métodos de una clase pertenecen a su interfaz y por tanto parece lógico que sean declarados como **públicos**, pero también es cierto que pueden darse casos en los que exista la necesidad de disponer de algunos métodos privados a la clase. Se trata de métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí forman parte de la interfaz. Ese tipo de métodos (de comprobación, de adaptación de formatos, de cálculos intermedios, etc.) suelen declararse como privados pues no son de interés (o no es apropiado que sean visibles) fuera del contexto del interior del objeto. Son métodos que no realizan las operaciones fundamentales que se tienen que hacer con ese objeto, como rotar (public), abrir (public), sino que sirven para algo intermedio, intrascendente.

Por ejemplo, supongamos que tenemos declarado el atributo DNI, y que sea necesario calcular la letra correspondiente a un determinado número de DNI o comprobar si una determinada combinación de número y letra forman un DNI válido. Este tipo de cálculos y comprobaciones podrían ser implementados en métodos privados de la clase (o al menos protegidos).

Los métodos con el modificador **protected** son accedidos solo por la propia clase y sus hijas (subclases).

Al igual que los atributos los métodos pueden ser de instancia y de clase.

Métodos de instancia. No llevan la palabra static. Pueden invocar a métodos y atributos de instancia a través de:

[puntero]. nombre del método

[puntero]. nombre del atributo

y a métodos y atributos de clase mediante:

[nombre de la clase]. nombre del método

[nombre de la clase]. nombre del atributo

Usamos el puntero o el nombre de la clase si llamamos al método o al atributo desde otra clase diferente a donde están definidos. Si se les llama desde la misma clase donde están definidos no se pone.

Métodos de clase. Tienen la palabra static. Un método **static** es un método cuya implementación es igual para todos los objetos de la clase.

Los métodos estáticos pueden usar e invocar a métodos y atributos estáticos, sin necesidad de tener un objeto de la clase que contiene el método y donde está definido el atributo creado. Esta invocación se realiza mediante:

[nombre de la clase]. nombre del método

[nombre de la clase]. nombre del atributo

Se usa el nombre de la clase cuando se les llama desde otra diferente a donde están definidos o implementados, y solo se pone el nombre del método o del atributo, si se les llama desde la misma clase donde están definidos o implementados.

No pueden sin embargo usar e invocar directamente a métodos o atributos de instancia, puesto que los atributos y métodos de instancia son propios de los objetos, y los métodos estáticos son independientes de los mismos. Para poder utilizar e invocar a un método o atributo de instancia, hay que crear un objeto de la clase donde estén definidos y llamarlos o usarlos mediante la referencia al objeto, siempre que ésta no esté declarada como un atributo.

puntero. nombre del método

puntero. nombre del atributo

Ejm:

```
public class Ejemplo {
     private static int numero;
     private int num=0;
   public  static boolean isPar()  {
        numero=10;  /* Desde un método estático se puede acceder a un
                       atributo estático*/
   }
  public static void incrementar() {
         num++;   //Esto dará un error
   }
}
```

Ejm:

```
public class Principal {
        Ejemplo obj;
        public Principal() {
                obj=new Ejemplo();
        }
        public void run() {
                obj.imprimir();
        }
        public static  void main(String[] args) {
                Principal p=new Principal();
                p.run();
        }  }
```

Los métodos que lleven el modificador de contenido final no podrán ser sobreescritos por ninguna clase que herede el método (se verá más adelante).

En Java el paso de parámetros es siempre por valor. En el caso de los tipos referenciados (por ejemplo los objetos) se pasa una referencia al objeto la cual no podrá ser cambiada, pero sí los elementos de su interior (atributos) a través de sus métodos, o por acceso directo si se trata de un miembro público.

- **Cuerpo del método**: contiene las sentencias que implementan el comportamiento del método (incluidas posibles sentencias de declaración de variables locales, aunque estas no deben coincidir con el nombre de un parámetro).

Los **elementos mínimos** que deben aparecer en la declaración de un método son:

- El tipo devuelto por el método.
- El nombre del método.
- Los paréntesis.
- El cuerpo del método entre llaves: { }

Ejemplo: En la clase Punto podemos encontrar el siguiente método:

```
int obtenerX (){
// Cuerpo del método
}
```

2.3.1- TIPOS DE MÉTODOS

a) **Get (obtener):** devuelve el contenido o valor de un atributo utilizando la sentencia return. Este tipo de métodos no recibe parámetros, ni hace cálculos, ni obtiene resultados intermedios o finales. Se pueden nombrar directamente con get seguido del nombre del atributo cuyo valor devuelven. **Ejm:** public int getEdad();

b) **Set (establecer):** nos permite modificar el contenido de un atributo de un objeto por el valor proporcionado a través de un parámetro de entrada. No devuelven ningún valor. **Ejm:** public void setEdad (int num);

c) **Constructores :** nos permiten crear el objeto.

d) **De propósito general:** realizan operaciones de carácter general.

e) **toString:** es un método sin parámetros. Nos permite mostrar la información completa de un objeto, es decir, el valor de sus atributos a través de una cadena de texto que será devuelta.

Este método se crea por defecto al crear el objeto, puesto que está definido en la clase Object (perteneciente al paquete java.lang) de la cual heredan todas las demás clases, y por tanto este método es heredado por lo que debemos sobreescribirlo.

Ejm:

```
public String toString(){
        String mensaje= "El valor de las coordenadas del punto son: "x=
        "+ x+ " y= "+y;
        return mensaje;
}
```

2.3.2- LA REFERENCIA THIS

This solo puede usarse en métodos de instancia (no static).

a) 'This' para referirse a los atributos de instancia de la clase actual

El uso de este operador puede resultar muy útil a la hora de evitar la ambigüedad que puede producirse entre el nombre de un parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo identificador (mismo nombre). En tales casos el parámetro "oculta" al atributo y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al atributo). La referencia this nos permite acceder a estos atributos ocultados por los parámetros mediante el operador punto (this.atributo).

Ejemplo:

```
public class Punto{
        private int x,y;
        public void establecerX (int x) {
            this.x= x;
       }
}
```

En el ejemplo utilizamos la referencia this, puesto que el nombre del parámetro del método coincide con el del atributo que se desea modificar.

En este caso es indispensable el uso de this, pues si no sería imposible saber en qué casos nos estamos refiriendo al parámetro x y en cuáles al atributo x. Para el compilador el identificador x será siempre el parámetro, pues ha "ocultado" al atributo.

En algunos casos puede resultar útil hacer uso de la referencia this aunque no sea necesario, cuando puede ayudar a mejorar la legibilidad del código.

b) This() en sobrecarga de constructores

Si se utiliza this() en un constructor debe ser la primera instrucción escrita en el mismo. Lo que hace es invocar a otro constructor que esté en la misma clase y que soporte los parámetros que le pasamos, si es que se los pasamos.

Ejemplo:

```
public class Rectangulo {
        private int x,y;
        private int alto, ancho;
        public Rectangulo() {
                this(0,0,1,1);
        }
        public Rectangulo(int alto, int ancho) {
                this(0,0,alto,ancho);
        }
        public Rectangulo(int x, int y, int alto, int ancho) {
                this.x=x;
                this.y=y;
                this.alto=alto;
                this.ancho=ancho;
        }
}
```

La única manera de invocar a un constructor puede ser dentro de otro. No podemos llamar a un constructor como si estuviéramos llamando a un método.

Ejemplo:

```
public class Horse{
        public Horse() { //constructor
        .....
        }
        public void metodo1() {
                Horse() //Llamada al constructor, ilegal
        }
}
```

La invocación recursiva al constructor no es válida en java.

3. OBJETOS

Una vez que se tiene implementada una clase con todos sus atributos y métodos ha llegado el momento de utilizarla, es decir, de instanciar objetos de esa clase e interaccionar con ellos.

Un objeto tiene una serie de características entre las que destacamos:

- **Estado**. Viene determinado por el valor de los atributos del objeto. **Ejm:** el estado de un coche puede ser: Fiat rojo 3 puertas.
- **Comportamiento**. Viene determinado por los métodos que usa dicho objeto. **Ejm:** cambiar_color.
- **Identidad**. Cada objeto es único y diferente de otro. Un objeto no es distinto de otro porque cambie su estado, sino porque tiene identidad. Así dos rotuladores pueden ser negros y estar vacíos, es decir, pueden tener los atributos iguales pero son dos entes distintos.

3.1.- DECLARACIÓN Y CREACIÓN DE UN OBJETO

La declaración de un objeto se realiza exactamente igual que la declaración de una variable de cualquier tipo:

<tipo> nombreVariable;

Donde tipo será alguna clase que se haya implementado, o bien alguna de las proporcionadas por la biblioteca de Java o por alguna otra biblioteca escrita por terceros.

Por ejemplo:

Punto p1;

Rectangulo r1, r2;

Coche cocheAntonio;

String palabra;

Las variables: p1, r1, r2, cocheAntonio, palabra, en realidad son referencias (también conocidas como punteros o direcciones de memoria) que apuntan (hacen “referencia”) a un objeto (una zona de memoria) de la clase indicada en la declaración.

Un objeto recién declarado no apunta a nada, se dice que es una referencia nula. Es decir, la variable existe y está preparada para guardar una dirección de memoria que será la zona donde se encuentre el objeto al que hará referencia, pero el objeto aún no existe (no ha sido creado o instanciado), por tanto se dice que apunta a un objeto inexistente.

Para que esa variable (referencia) apunte realmente a un objeto (contenga una referencia o dirección de memoria que apunte a una zona de memoria en la que se ha reservado espacio para un objeto), es necesario crear o instanciar el objeto utilizando el **operador new**, el cual tiene la siguiente sintaxis:

nombreObjeto= new <ConstructorClase> ([listaParametros]);

Un objeto puede ser declarado e instanciado en la misma línea (objeto e instancia son términos equivalentes).

Punto p1= new Punto ();

De la tarea de reservar memoria para la estructura del objeto (sus atributos más alguna otra información de carácter interno para el entorno de ejecución) se encarga el propio entorno de ejecución de Java. Es decir, el entorno sabrá que tiene que realizar una serie de tareas (solicitud de una zona de memoria disponible, reserva de memoria para los atributos, enlace de la variable objeto a esa zona, etc.) y se pondrá rápidamente a desempeñarlas.

Algunos ejemplos de instanciación o creación de objetos podrían ser:

p1= new Punto ();

r1= new Rectangulo ();

cocheAntonio= new Coche();

3.1.1- CONSTRUCTOR DE UNA CLASE

El constructor de una clase es una especie de método que tiene toda clase y se invoca de manera "automática" al crear el objeto. Su nombre coincide con el de la clase a la que pertenece y no devuelve ningún valor tras su ejecución. Es quien se encarga de inicializar los atributos (es su función principal, aunque puede contener código). Dado que el constructor es como un método más de la clase podrá tener también su lista de parámetros como tienen todos los métodos.

Los atributos se inicializan en 3 etapas:

1. El sistema asigna a cada atributo su valor por defecto (0 para lo numéricos, null para las referencias, false para los boolean, etc). Los atributos son inicializados por defecto, las variables locales de los métodos no se inicializan por defecto.
2. Si la declaración del atributo contiene un inicializador, éste es evaluado y asignado.
3. Se ejecuta el constructor, que puede cambiar el valor del atributo que se desee.

Es recomendable que en un constructor se inicialicen todos los atributos de la clase aunque su valor vaya a ser nulo o vacío, para no trabajar con malas prácticas de programación.

Cuando se escribe el código de una clase no es necesario implementar el constructor si no se quiere hacer. Java se encarga de dotar de un constructor por omisión (también conocido como constructor por defecto) a toda clase, puesto que toda clase debe tener al menos un constructor. El constructor por defecto no tiene parámetros, ni cuerpo, es un constructor vacío que no hace nada. No se ve en el código de la clase, lo incluirá el compilador de Java al compilarla si descubre que no se ha creado ningún método constructor para esa clase (queda incluido en el .class).

En el caso de que incluyamos un constructor personalizado a una clase, el compilador ya no incluirá el constructor por defecto. Si queremos que la clase tenga también un constructor sin parámetros tendremos que escribir su código. En este caso se trata de una redefinición del constructor por defecto.

Un ejemplo de constructor podría ser:

```
package ejemplo;
public class Punto {
        private int x,y;
        public Punto (int x, int y) {
            this.x= x;
            this.y= y;
        }
}
 class Principal {
        public static void main (String [ ] args) {
                    Punto p1= new Punto (10, 7);
        }
}
```

En este caso el constructor recibe dos parámetros (no es el constructor por defecto). Asigna sendos valores iniciales a los atributos x e y.

En la definición de un constructor se indica:

- El tipo de acceso.
- El nombre de la clase (el nombre de un constructor es siempre el nombre de la propia clase).
- La lista de parámetros que puede aceptar (si hay).
- Si lanza o no excepciones (ya se estudiarán más adelante).
- El cuerpo del constructor (un bloque de código como el de cualquier método).

La estructura de un constructor es similar a la de cualquier método, con las excepciones de que **no tiene tipo de dato devuelto** (no se escribe ni tan siquiera void) y que **el nombre del constructor debe ser obligatoriamente el nombre de la clase**.

3.1.2- CONSTRUCTOR DE COPIA

Durante el proceso de creación de un objeto se pueden generar objetos exactamente iguales (basados en la misma clase), que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase que además de generar un objeto nuevo que lo haga con los mismos valores que tenga otro objeto ya existente. Es decir, algo así como si se pudiera **clonar** el objeto tantas veces como haga falta. A este tipo de mecanismo se le suele llamar **constructor copia** o **constructor de copia**.

Un constructor copia es por tanto un constructor, pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este constructor revisa cada uno de los atributos del objeto recibido como parámetro, y copia todos sus valores en los atributos del objeto que se está creando en ese momento.

Un ejemplo de constructor copia podría ser:

```
package ejemplo;

public class Punto {
    private int x,y;
    public Punto (int x, int y) {
        this.x= x;
        this.y= y;
    }
```

```
        public Punto (Punto p) {
            this.x= p.x;
            this.y= p.y;
        }
}
 class Principal {
        public static void main (String [ ] args) {
              Punto p1= new Punto (10, 7);
              Punto p2= new Punto (p1);
        }
}
```

En este caso el constructor recibe como parámetro un objeto del mismo tipo que el que va a ser creado (Punto), inspecciona el valor de los atributos (x e y) y los reproduce en los atributos del objeto en proceso de construcción. Así pues p2 se crea a partir de los valores del objeto p1. Los atributos x,y del objeto p2 tendrán como valor 10 y 7 respectivamente.

3.2.-MANIPULACIÓN DE UN OBJETO. UTILIZACIÓN DE MÉTODOS Y ATRIBUTOS

Una vez que un objeto ha sido declarado y creado se puede decir que el objeto existe en el entorno de ejecución, y por tanto que puede ser manipulado como un objeto más en el programa haciendo uso de sus atributos y sus métodos.

Para acceder a un miembro de un objeto desde una clase diferente a la de su implementación o declaración, se utiliza el nombre de la variable objeto, seguida del operador punto (.) del siguiente modo:

<nombreVbleObjeto>.<nombreMiembro>

Donde <nombreMiembro> será el nombre de algún miembro del objeto (atributo o método) al cual se quiere tener acceso.

Sin embargo cuando se quiera acceder desde la misma clase a la de su implementación o declaración, pueden ser referenciados directamente.

Ejm:

```
public class Ejemplo {
 private double largo, ancho;
   public double area() {
       return largo*ancho;
    }
}
```

```
public class Resultado {
    public static void main(String[ ] args) {
         Ejemplo ob=new Ejemplo();
         System.out.println(ob.area());
     }
}
```

4. SOBRECARGA

El lenguaje Java soporta la característica conocida como **sobrecarga de métodos**. Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre (cuando existen varias formas de invocar a un método con el mismo nombre pero con distintos parámetros estamos hablando de métodos sobrecargados). La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre será mediante la lista de parámetros del método. Si el método tiene una lista de parámetros diferente (colocados en distinto orden, tipos diferentes o distinto número de ellos), será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos o más métodos con el mismo nombre en la misma clase. Cuando se llama a un método sobrecargado java busca una versión del método cuyos parámetros coincidan con los argumentos utilizados en la llamada al método. Sin embargo esta coincidencia no tiene por qué ser exacta, en algunos casos las conversiones automáticas juegan un papel importante en la resolución de la sobrecarga.

Ejm:

obj.valores (15f,24f);

Puede llamar al método: **public void valores (double largo, double ancho)**

La conversión automática de tipo solo se aplica si no existe una coincidencia exacta entre argumentos y parámetros.

Debemos tener en cuenta que el tipo devuelto por el método no es considerado a la hora de identificar un método. Por tanto un tipo devuelto diferente no es suficiente para distinguir un método de otro. Es decir, no podríamos definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El compilador produciría un error de duplicidad en el nombre del método y no lo permitiría. El compilador distingue a los métodos por su firma o signatura: nombre del método, parámetros (número, orden y tipo) y clase a la que pertenece. No puede haber dos métodos con la misma signatura.

Es conveniente no abusar de la sobrecarga de métodos y utilizarla con cierta moderación (cuando realmente puede beneficiar su uso), dado que podría hacer el código menos legible.

Los constructores pueden ser sobrecargados, de esta forma un objeto puede inicializarse de varias formas. La sobrecarga de constructores es la norma y no una excepción.

5. CADENAS DE CARACTERES

Entre las clases que sirven para manejar cadenas de caracteres destacamos:

- La clase String
- La clase StringBuffer

5.1.- La clase String

Los String no forman parte de los tipos primitivos de java, sino que existe una clase String dentro del paquete java.lang (java.lang.String).

La clase String permite manejar cadenas de caracteres cuyo tamaño y contenido es fijo, no se altera a lo largo de la vida del programa, es decir, esta clase permite el manejo de cadenas de caracteres no modificables. Así pues, si se realiza una concatenación de un String con otro ya existente, se creará uno nuevo con el contenido resultante de la operación. Por tanto, una instancia u objeto String es un puntero o dirección de memoria que apunta hacia una estructura de datos, que contiene el conjunto de caracteres que define un String.

De una manera informal podemos definirlo como una serie de caracteres delimitados por comillas dobles. Las comillas simples pueden formar parte del contenido del String.

Ejm: String frase= "el chico 'pesado' vino a verme"

Los Strings u objetos de la clase *String* se pueden crear explícitamente o implícitamente.

Para crear un String implícitamente basta poner una cadena de caracteres entre comillas dobles. Por ejemplo:

```
System.out.println("El primer programa");
```

Java crea un objeto de la clase *String* automáticamente.

Para crear un String explícitamente podemos utilizar los siguientes constructores:

1. **String (String str)**

   Ejm: String str=new String("El primer programa").

   Se crea la cadena "El primer programa":

   También se puede realizar una asignación directa a una variable declarada de tipo String.

   String str ="El primer programa";

2. **String()**

   Ejm: String str=new String(); //se inicializa la cadena a “”

   Se crea una cadena vacía.

   También se puede hacer mediante asignación:

   String str="";

Un String vacio es aquél que no contiene caracteres, pero es un objeto de la clase *String*.

Hay que recordar que los espacios cuentan. No es lo mismo una cadena de longitud cero o cadena vacía representada por dos comillas sin espacio entre ellas, que una cadena que contenga un espacio (representada por dos comillas que contengan un espacio entre ellas), cuya longitud o número de caracteres es 1.

Un String declarado pero sin definir todavía no apunta hacia ninguna estructura de datos.

Ejm: String vacio;

Para indicar que no apunta hacia ninguna estructura de datos conviene inicializarlo a null.

Ejm: String vacio=null;

Métodos principales de la clase String

Una vez creado un objeto de la clase *String,* podemos obtener información y manipularlo a través de sus métodos. La signatura de los mismos podemos consultarla en la documentación del API (interfaz de programación de aplicaciones) de java.

Entre los métodos de la clase String destacamos:

● **int length():** devuelve la longitud de la cadena de caracteres, esto es, número de caracteres que tiene incluyendo espacios en blanco. La longitud siempre es una unidad mayor que el índice asociado al último carácter de la cadena (el primer carácter está en la posición cero).

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

int longitud=str.length();
```

● **char charAt (int posicion):** devuelve el carácter de la cadena colocado en la posición indicada en el parámetro. El acceso a una posición fuera de los límites de la cadena genera un error.

Ej: System.out.println(cadena.charAt(0)); //devuelve el carácter que está en la primera posición de la cadena.

Es erróneo usar charAt(length()), ya que aunque la cadena tiene length() caracteres, al empezar en la posición 0 el último se encuentra en length() -1.

Comparación de Strings

La comparación de Strings nos da la oportunidad de distinguir entre el operador lógico == y el método *equals* de la clase *String*.

El operador lógico "==" funciona de manera diferente. Así por ejemplo:

1.
```
String s1= new String ("campusMVP");
String s2= new String ("campusMVP");
System.out.println (s1==s2); //Visualiza false
```

2.
```
String s1= "campusMVP";
String s2= new String ("campusMVP");
System.out.println (s1==s2); //Visualiza false
```

3. String s1= "campusMVP";

   String s2= "campusMVP";

   System.out.println (s1==s2); //Visualiza true

4. String s1= "campusMVP";

   String s2= s1;

   System.out.println (s1==s2); //Visualiza true

¿A qué se debe que la salida sea una u otra? A su codificación en memoria, que puede variar dependiendo si las cadenas se crean mediante new o por asignación.

Las cadenas creadas por asignación se almacenan en un espacio especial de la memoria denominado repositorio de cadenas (String pool), que va a contener una única copia de las cadenas.

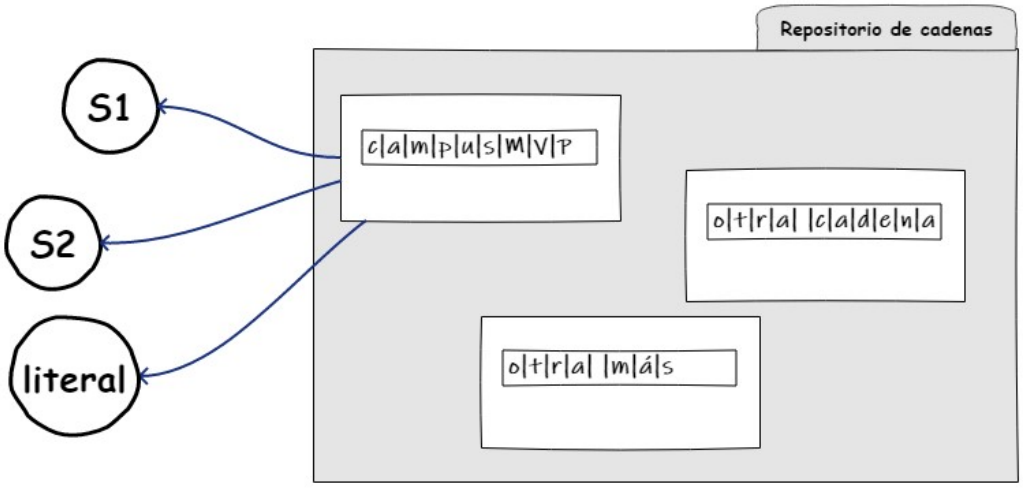
El repositorio funciona de la siguiente forma: cuando se crea una cadena por asignación, la máquina virtual de java examina el repositorio a ver si hay almacenada ya una cadena con igual contenido. Si no la hay, la almacena en una dirección de memoria determinada, y si ya está almacenada, una vez localizada devuelve la dirección de memoria al objeto ya almacenado.

Ejm:

String s1= "campusMVP";

String s2= "campusMVP";

String literal= "campusMVP";



Funciona de igual forma con la combinación de cadenas.

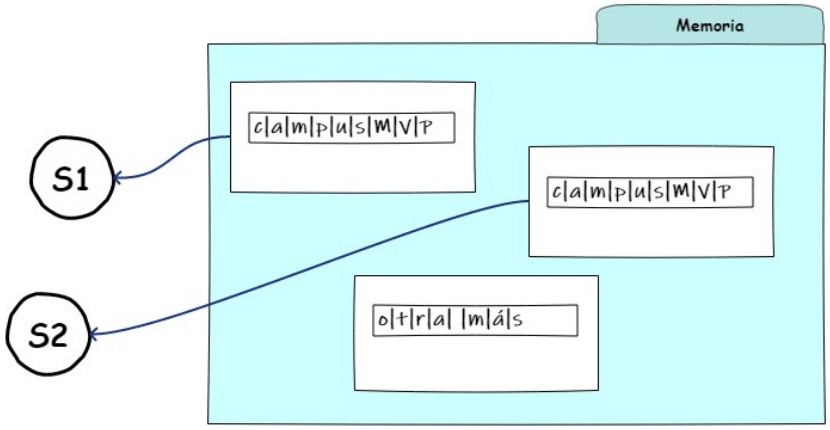
Ejm:

```
String s1= "campusMVP";
String s2= "campus"+"MVP";
System.out.println(s1==s2); //Visualiza true
```

Sin embargo cuando se crean cadenas usando el new, no se almacenan en el repositorio, si no que se reserva espacio en memoria principal para cada cadena devolviendo su dirección.

Ejm:

```
String s1= new String ("campusMVP");
String s2= new String ("campusMVP");
```



Los métodos que operan con cadenas, generan nuevas cadenas a partir de la original y por tanto se trata de objetos diferentes.

Ejm:

```
String s1= "campusMVP";
String s2= "campusMVP";
System.out.println (s1.toLowerCase() == s2.toLowerCase());
/*Visualizará false, puesto que se crean dos cadenas idénticas pero
representadas por dos objetos diferentes*/
```

El método, **boolean equals (String str),** investiga si dos cadenas tienen los mismos caracteres y en el mismo orden. Si es así devuelve true, y si no false.

```
String str="El lenguaje Java";
boolean resultado=str.equals("El lenguaje Java");
```

La variable *resultado* tomará el valor true.

● **boolean equalsIgnoreCase (String str):** investiga si dos cadenas tienen los mismos caracteres y en el mismo orden sin tener en cuenta las mayúsculas. Si es así devuelve true, y si no false (como equals pero sin distinguir mayúsculas y minúsculas).

• **int compareTo (String str):** devuelve un entero menor que cero si el String de referencia es menor (en orden alfabético) que el String pasado por parámetro, cero si son iguales, y mayor que cero si el String de referencia es mayor que el pasado por parámetro (el entero que devuelve es la diferencia entre los primeros caracteres no iguales entre ambas cadenas).

```
String str ="Tomás";
int resultado=str.compareTo("Alberto");
```

La variable entera *resultado* tomará un valor mayor que cero, ya que Tomás está después de Alberto en orden alfabético.

```
String str="Alberto";
int resultado=str.compareTo("Tomás");
```

La variable entera *resultado* tomará un valor menor que cero, ya que Alberto está antes que Tomás en orden alfabético.

Otros métodos que nos permiten comparar cadenas son:

● **boolean startsWith (String str):** comprueba si el String de referencia comienza con los caracteres especificados en la cadena pasada como parámetro. Devuelve true o false según que el String comience o no por dichos caracteres.

```
String str ="El primer programa";
boolean resultado=str.startsWith("El");
```

En este ejemplo la variable resultado tomará el valor true.

● **boolean startsWith (String str, int indice):** nos permite conocer si el String de referencia comienza con los caracteres especificados en la cadena pasada como primer parámetro, a partir de una determinada posición especificada como segundo parámetro. Devuelve true o false según que el String comience o no por dichos caracteres.

● **boolean endsWith (String str):** devuelve true si el String de referencia finaliza con los caracteres especificados en la cadena pasada por parámetro, false en caso contrario.

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

boolean resultado=str.endsWith("programa");
```

Extraer un substring de un String

En muchas ocasiones es necesario extraer una porción de cadena o substring de un String dado. Para este propósito usaremos el método substring que puede ser invocado de dos formas:

1. String substring (int comienzo): nos devuelve el substring formado por los caracteres situados desde un posición determinada pasada por parámetro (incluida), hasta el final del String.

Ejm:
```
String str="El lenguaje Java";
String subStr=str.substring(12);
```
Se obtendrá el substring "Java".

2. String substring (int comienzo, int final): nos devuelve el substring formado por los caracteres que van desde la posición comienzo a (final-1) (incluidas ambas), pasadas por parámetro.

Si el número de caracteres de la cadena es menor que el índice de comienzo se produce un error, al igual que si el índice final sobrepasa la longitud del String. Podríamos evitar este tipo de errores si controlamos cuál es la longitud o número de caracteres de las cadenas con las que trabajamos a través del método length.

Ejm:
```
String str="El lenguaje Java";
String subStr =str.substring(3, 11);
```

Obtendrá el substring "lenguaje". Hay que recordar que las posiciones se empiezan a contar desde cero.

Ejm:

"coco".substring(2,2). Devuelve "" (cadena vacía).

"coco".substring(7,3). Devuelve un error al no existir el carácter situado en la posición 7 dentro de la cadena.

Búsqueda dentro de cadenas

● **int indexOf (String str):** devuelve la posición en la que aparece por primera vez el String que se le pasa como parámetro, comenzando la búsqueda desde el principio de la cadena. Si el String no aparece devuelve –1.

Ejm: String frase="Tenemos que decir muchas cosas";

posicion=frase.indexOf ("que");

Resultado: 8

● **int indexOf (String str, int posicion):** devuelve la posición en la que aparece por primera vez el String que se le pasa como primer parámetro, comenzando la búsqueda a partir de la posición especificada en el segundo parámetro. Si la posición a partir de la que se inicia la búsqueda no existe o el String no aparece, devuelve –1.

Ejm:

String frase="Tenemos que decir muchas cosas";

int posicion=frase.indexOf ("que",3);

Resultado: posición: 8 (T e n e m o s q u e)
0 1 2 3 4 5 6 7 8

● **int indexOf (char carácter):** devuelve la posición en la que aparece por primera vez el carácter pasado como parámetro, comenzando la búsqueda desde el principio de la cadena. Si no se encuentra el carácter devuelve el valor –1.

Ejm: String str="El primer programa";

int pos=str.indexOf('p');

Resultado: 3

● **int indexOf (char carácter, int comienzo):** devuelve la posición en el que aparece por primera vez el carácter que se le pasa como primer parámetro, comenzando la búsqueda a partir de la posición especificada en el segundo parámetro. Si no se encuentra el carácter devuelve el valor –1.

Ejm: String str="El primer programa";

pos=str.indexOf('p', 4);

Resultado: 10

● **int lastIndexOf (char carácter):** devuelve la posición en la que aparece por última vez el carácter pasado como parámetro, comenzando la búsqueda desde el principio de la cadena.

Ejm: String s1="this is indes of example"; //Hay dos 's' en la frase

int posicion=s1.lastIndexOf('s');//retorna la posición de la última 's'

System.*out*.println(posicion);

Resultado: 12

● **int lastIndexOf (int carácter, int posicion):** devuelve la posición de la última aparición del carácter pasado como primer parámetro, comenzando a buscar desde el principio de la cadena hasta la posición indicada en el segundo parámetro.

Ejm: String s1="this is index of example";

int index1=s1.lastIndexOf('s',9);

System.*out*.println(index1);

Resultado: 6

● **int lastIndexOf (String str):** devuelve la posición de la última aparición del String que se pasa por parámetro, comenzando a buscar desde el principio de la cadena.

Ejm: String s1="this is index of example";

int index1=s1.lastIndexOf("is");

System.*out*.println(index1);

Resultado: 5

● **int lastIndexOf (String str, int fin):** devuelve la posición de la última aparición del String pasado como parámetro, comenzando a buscar desde el principio de la cadena hasta la posición indicada en el segundo parámetro.

Modificación de cadenas

Las cadenas de caracteres no modifican su valor pero se crean copias de la cadena con su valor modificado.

Los siguientes métodos permiten modificar cadenas:

● **String toLowerCase():** devuelve un nuevo String convirtiendo todos los caracteres del String sobre el que se aplica el método a minúsculas.

● **String toUpperCase():** devuelve un nuevo String convirtiendo todos los caracteres del String sobre el que se aplica el método a mayúsculas.

● **String replace (char viejoChar, char nuevoChar):** cambia en el String de referencia el carácter especificado como primer parámetro por el especificado en el segundo, generando un nuevo String.

• **String trim():** devuelve un String pero sin espacios en blanco ni al principio ni al final de la cadena. No elimina los espacios en blanco situados entre las palabras.

Ejm:
```
String str=" 12 ";
String str1=str.trim();
```

Convertir un número a String

Convierte valores de tipo primitivos de datos (int, long, float, double, char, boolean) a String.

Podemos usar:

1. El método estático valueOf:

- **String valueOf(int i):** convierte un entero de 32 bits a cadena.

Ejm:
```
int valor=10;
String str=String.valueOf(valor);
```

- **String valueOf (long l):** convierte un entero de 64 bits a cadena.
- **String valueOf (float f):** convierte un flotante de 32 bits a cadena.
- **String valueOf (double d):** convierte un double de 64 bits a cadena.

- **String valueOf (char c):** convierte un carácter en cadena.
- **String valueOf (boolean b):** convierte un boolean a cadena.

2. A través de las clases de envoltura y el método toString()

Las clases de envoltura (Integer, Long, Float, Double, Character y Boolean) pertenecientes al paquete java.lang, nos permiten encapsular los tipos primitivos de datos para crear un objeto y así utilizar sus métodos correspondientes, entre ellos el método toString() genérico para cualquier clase, lo que nos va a permitir convertir un objeto en una cadena de caracteres.

- **String Integer.toString(int i)**

```
Ejm:  int valor=10;
      String str= Integer.toString(valor);
```

- **String Float.toString(float f)**
- **String Double.toString(double d)**
- **String Character.toString(char c)**
- **String Boolean.toString(boolean b)**

Convertir un String en número

Podemos usar:

1. Los métodos estáticos parseInt, parseLong, parseFloat, parseDouble, o parseBoolean de las clases de envoltura, nos permiten realizar la conversión de String a cada uno de los tipos básicos correspondientes.

- **int Integer.parseInt(String str)**

```
Ejm:  String str="10";
      int numero=  Integer.parseInt(str);
```

Si en la cadena hay espacios en blanco al comienzo o al final de la misma, debemos quitarlos primero mediante el método trim() antes de convertirla en número.

```
Ejm:  String str="10";
      int numero=  Integer.parseInt(str.trim());
```

- **long Long.parseLong(String str)**
- **float Float.parseFloat(String str)**
- **double Double.parseDouble(String str)**
- **boolean Boolean.parseBoolean(String str)**

2. A través de las clases de envoltura junto con los métodos intValue(), longValue(), floatValue(), doubleValue(), CharValue () y booleanValue().

- **int new Integer (String str).intValue()**

Ejm: String str="10";

int numero= new Integer(str).intValue();

- **long new Long (String str).longValue()**
- **float new Float (String str).floatValue()**
- **double new Double (String str).doubleValue()**
- **char new Character (String str).charValue()**
- **boolean new Boolean (String str).booleanValue()**

Concatenación de cadenas

1. Operadores (+) (+=)

Permiten unir o concatenar cadenas de caracteres. Se pueden usar con literales (“……”).

Ejm:

```
int result=3;
System.out.println (“resultado” + result);
```

En este caso antes de hacer la concatenación se hace una conversión implícita a String.

También se pueden usar los operadores con objetos propiamente dichos.

Ejm:

```
String frase1=”hola”;
String frase2=”mundo”;
String resultado=frase1+frase2;
frase1+=frase2;
```

```
System.out.println (resultado+" "+frase1); //holamundo holamundo
System.out.println ("hola"+frase2); // holamundo
```

2. El método, **String concat (String str)**, nos permite concatenar el String de referencia al que se pasa por parámetro.

Ejm:

```
String frase1=”Tomás”;
String frase2=”Polac”;
String posic=frase1.concat(frase2);
```

Formateado de cadenas

En java se les puede dar formato a las cadenas a través del método estático format de la clase String, o a través del método printf de la clase PrintStream.

Cualquiera de estos métodos consta de dos argumentos:

1. El denominado **cadena de formato o formato de salida**: es una cadena que especifica cómo será el formato de salida. En ella se puede mezclar texto normal con uno o varios especificadores de formato, que indican cómo se deben formatear el o los argumentos que hay después de la cadena de formato.

   Los especificadores de formato comienzan siempre por **%** (carácter de escape, que sirve para indicar que lo que hay a continuación no es texto normal sino algo especial), seguido por un carácter que indica la conversión a realizar, por ejemplo **%d**. Dependiendo del tipo de dato podemos usar unas conversiones u otras.

2. **Lista de argumentos:** variable/s cuyos valores se visualizarán siguiendo el formato especificado. Tiene que haber tantos argumentos como especificadores de formato en la cadena de formato.

Listado de conversiones más utilizada y ejemplos

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplos
Número decimal de simple precisión	**"%f"**	Flotantes simples Dobles	float i=10.5f; System.out.println (String.format("Resultado: %f", i)); System.out.printf("Resultado:%f", i); **Resultado: 10,500000** Resultado sin formatear: 10,5
Número en notación científica o decimal (lo más corto)	**"%g"** o **"%G"**	Flotantes simples Dobles. El número se mostrará como decimal o en notación científica dependiendo de lo que sea más corto.	double i=10.5; System.out.println (String.format("Resultado: %g", i)); System.out.printf("Resultado:%g", i); **Resultado: 10,5000** Resultado sin formatear: 10,5

Valor lógico o booleano	**"%b"** o **"%B"**	Boolean
		(cuando se usan otros tipos de datos siempre lo formateará escribiendo true)
			boolean v=true;
			System.out.println (String.format("Resultado: %b",v));
			
			Resultado: true
			Resultado sin formatear: true
			
			float i=10.5f;
			System.out.println (String.format("Resultado: %b", i));
			System.out.printf("Resultado:%b", i);
			Resultado: true
			Resultado sin formatear: 10,5

Tipo de conversión	**Especificación de formato**	**Tipos de datos aplicables**	**Ejemplos**

Número en notación científica	**"%e"** o **"%E"**	Flotantes simples
		Dobles
			double i=10.5;
			String d= String.format("Resultado: %E", i);
			System.out.println (d); //
			System.out.println(String.format ("Resultado: %E", i));
			
			Resultado:1.050000e+01
			Resultado sin formatear: 10,5

Cadena caracteres	**"%s"** o **"%S"**	Cualquiera. Se convertirá el objeto a cadena si es posible	String c="hola mundo"; System.out.println (String.format("Resultado: %s", c)); **Resultado: hola mundo** float i=10.5f; System.out.println (String.format("Resultado: %s", i)); System.out.printf("Resultado:%s", i); **Resultado:1,5 //**Es una cadena **Resultado sin formatear: 10,5 //** Es un número
Número entero	**"%d"**	Entero	int i=10; System.out.println (String.format("Resultado: %d", i)); System.out.printf("Resultado:%d", i); **Resultado:10** Resultado sin formatear: 10
Carácter ASCII	**"%c"**	Entero Carácter	char i='a'; System.out.println (String.format("Resultado: %c", i)); System.out.printf("Resultado:%c", i); **Resultado:a** Resultado sin formatear: a
Tipo de conversión	**Especificación de formato**	**Tipos de datos aplicables**	**Ejemplos**
Hexadecimal sin signo	**"%x"** o **"%X"**	Entero	int i=20; System.out.println (String.format("Resultado: %x", i)); System.out.printf("Resultado:%x", i); **Resultado:14** Resultado sin formatear: 20
Octal sin signo	**"%o"**	Entero	int i=20; System.out.println (String.format("Resultado: %o", i)); **Resultado:24** Resultado sin formatear: 20

Imprime el símbolo %	**"%%"**	Cualquiera	int i=20; System.out.println(String.format ("Resultado: %d%%", i)); **Resultado:20%** Resultado sin formatear: 20

Modificadores que se pueden aplicar a las conversiones

Se pueden utilizar modificadores para ajustar como queremos que sea la salida. Se sitúan entre el carácter de escape (%) y la letra que indica el tipo de conversión (d,f,g,etc).

Sintaxis general:

% [Indices de argumento] [Flag/s] [Ancho] [.Precisión] Conversión

Los corchetes en la notación significa que es opcional.

Ejemplos:

% Ancho Conversión

Nos permite especificar el número de caracteres que tendrá como mínimo la conversión. Si el dato a mostrar no llega a ese ancho en caracteres, se rellenará con espacios en blanco a la izquierda.

Si el dato a mostrar sobrepasa el ancho especificado, se mostrará el dato ignorando el ancho especificado.

La salida se justifica a la izquierda si no se especifica el ancho o es menor que el dato en sí, en caso contrario se justifica a la derecha.

Este tipo de modificador se puede usar con cualquier conversión.

Ejm:

```
System.out.println(String.format("%5d",10));
System.out.printf("%5d", 10);
```

Salida: 10 //Se muestra el 10 utilizando 5 espacios (se rellena por la izquierda con 3 espacios en blanco)

Ejm:

```
int num=12;
int num2=12345;
System.out.println(String.format("%d\n%5d\n",num2,num));
```

Salida: 12345
12

Es útil cuando se van a imprimir datos en columnas.

Ejm:

```
int num=2000;
System.out.println(String.format("%2d",num));
```

Salida: 2000

% .Precisión Conversión **% Ancho .Precisión Conversión**

.Precisión: nos permite indicar la precisión de la conversión. La precisión es el tamaño de la parte decimal para números reales y el número de caracteres a imprimir para cadenas de texto.

Ejm:

```
float precio=3.3f;
System.out.println (String.format("El precio es: %.2f ", precio));
System.out.printf("El precio es: %.2f ", precio);
```

Salida: El precio es: 3,30 // El número se completó con un cero por la derecha puesto que solo tenía un decimal.

Ejm:

```
System.out.println(String.format("%8.3f",4.2f));
```

Salida: 4,200 //El número se completó con 3 espacios en blanco por la izquierda. La coma cuenta como un carácter más.

Ejm:

```
System.out.println(String.format("%3.3f",4.2f));
```

Salida:4,200 // Se completó la precisión ignorando el ancho especificado

Ejm:

```
String np="Lavadora";
int u=10;
float ppu = 302.4f;
float p=u*ppu;
System.out.println(String.format("Producto: %s; Unidades: %d; Precio por unidad: %.2f €; Total: %.2f €", np, u, ppu, p));
```

Salida: Producto:Lavadora; Unidades:10; Precio por unidad:302,40 €; Total: 3024,00 €

Ejm:

```
System.out.println(String.format("Color: %s, Número: %d, Real: %5.2f", "rojo", 1234567,3.14));

System.out.printf(("Color: %s, Número: %d, Real: %5.2f ",  "rojo", 1234567,3.14));
```

Salida: Color: rojo, Número: 1234567, Real:3,14

Si en la cadena de formato aparecen varios especificadores, los argumentos a incluir se toman en el mismo orden en el que aparecen. No se comprueba que el número de especificadores en la cadena de formato y el número de argumentos sea el mismo. En caso de que no coincidan se producirá un error de ejecución.

%Flag Conversión **% Flags Ancho .Precisión Conversión**

Flag	Tipo de datos aplicables	Descripción
0	Float Double Entero	Se debe especificar el ancho. Permite rellenar con ceros a la izquierda hasta el valor especificado en el ancho.
+	Float Double Entero	Imprime el signo de un número cuando es positivo.

No son excluyentes entre sí los flags.

Ejm:

```
int num=12;
double i=3.1417;
System.out.println(String.format("%+d",num));
System.out.println(String.format("%0+7.2f",i));
```

Salida:

```
+12
+003,14
```

% Indices de argumento Conversión

Cuando el orden de los argumentos es un poco complicado porque se reutilizan varias veces en la cadena de formato, se puede recurrir a los índices de argumento. Se trata de especificar la posición del argumento a utilizar (el primer argumento sería el 1), seguido por el símbolo del dólar ("**$**"). El índice se ubicaría al comienzo del especificador de formato después del porcentaje.

Ejm:

```
int i=10;
int j=20;
String d=String.format("%1$d multiplicado por %2$d (%1$d x %2$d) es
%3$d",i,j,i*j);

System.out.println(d);
```

Salida: 10 multiplicado por 20 (10 x 20) es 200

Los índices de argumento se pueden usar con todas las conversiones, siendo compatibles con otros modificadores de formato (incluida la precisión).

5.2.- La clase StringBuffer

En java como ya comentamos la clase String nos permite crear objetos inmutables, es decir de tamaño y contenido fijo, lo que significa que cada vez que se quiera modificar un String se debe crear un nuevo objeto donde queda plasmada esa modificación. Esto conlleva un alto consumo de memoria, de ahí que cuando implementemos un programa que realice muchas operaciones con cadenas, o si el contenido de las mismas va a ser modificado en un programa, es necesario optimizar el uso de la memoria.

Java proporciona la clase StringBuilder y la clase StringBuffer (pertenecientes al paquete de java.lang), que son mutables, y por tanto permiten una mayor optimización de la memoria. Vamos a centrarnos en la clase StringBuffer al estar pensada para aplicaciones multi-hilo.

En la clase StringBuffer se debe tener en cuenta no sólo la longitud de la cadena de caracteres que contiene (length) sino también la capacidad total (capacity), que indica cuántos caracteres puede almacenar sin solicitar más memoria; si se excede ésta durante el uso el sistema automáticamente reservará más espacio, pero eso requiere más trabajo para el sistema.

Al igual que en la clase String, el primer carácter de una cadena StringBuffer se encuentra almacenado en la posición cero.

Constructores

Existen distintos constructores para crear objetos de tipo StringBuffer:

- **StringBuffer():** crea un StringBuffer con capacidad para 16 caracteres.
- **StringBuffer(int n):** crea un StringBuffer con capacidad para n caracteres.
- **StringBuffer(String s):** crea un StringBuffer con capacidad para s.length()+16 caracteres, y cuyo contenido será la cadena s.

Ejm:

```
StringBuffer uno=new StringBuffer(20); //crea un StringBuffer vacío de longitud 20.
StringBuffer dos=new StringBuffer("Ejemplo");
StringBuffer tres=new StringBuffer(); // se inicializa a “”
```

Métodos relacionados con el tamaño

El cambio de tamaño de un objeto StringBuffer necesita varios métodos específicos para manipular el tamaño de las cadenas.

- **int length():** devuelve el número de caracteres de la cadena.
- **int capacity():** devuelve el número máximo de caracteres que puede contener la cadena.

  Ejm:

```
StringBuffer str;
str=new StringBuffer();
System.out.println(str.length()+" "+str.capacity());
```

- **void setLength (int i):** establece el tamaño que tendrá la cadena pasándolo por parámetro. Si el número es menor al de caracteres existentes en la misma, se trunca la cadena.

Ejm:

```
StringBuffer str;
str=new StringBuffer("cadena caracteres");
str.setLength(3);
System.out.println(str);
```

Salida: cad

- **void ensureCapacity(int num):** reserva el espacio de memoria para almacenar num caracteres. Si la capacidad original es menor que num, el método coloca como capacidad la mayor de las cantidades siguientes: num o el doble de la capacidad original más 2, es decir, se asigna una nueva y mayor capacidad; en caso contrario mantiene la capacidad original.

Ejm:

```
StringBuffer s2;
s2=new StringBuffer(5);
s2.ensureCapacity(8);
//Establece la capacidad a 5*2+2 que es mayor que 8
System.out.println(s2.capacity()); //imprime 12
```

Métodos relacionados con los caracteres

- **char charAt(int pos):** obtiene el carácter almacenado en la posición pos.

Ejm:

```
StringBuffer s2;
s2=new StringBuffer("cadena");
System.out.println(s2.charAt(4)); //imprime 'n'
```

- **void setCharAt(int pos, char c):** modifica el carácter que está en la posición pos, por el carácter 'c'.

Ejm:

```
StringBuffer s2;
s2=new StringBuffer("cadena");
s2.setCharAt(4, 'x');
System.out.println(s2); //imprime cadexa
```

- **StringBuffer deleteCharAt(int pos):** devuelve el StringBuffer original después de eliminar el carácter almacenado en la posición pos y reducir su longitud.

Ejm:

```
StringBuffer s2;
s2=new StringBuffer("cadena");
s2.deleteCharAt(2);
System.out.println(s2);//imprime caena
```

Búsqueda dentro de cadenas

• **int indexOf (String str):** devuelve la posición en la que aparece por primera vez la cadena que se le pasa como parámetro, comenzando la búsqueda desde el principio de la misma. Si el String no aparece devuelve –1.

Ejm:

```
StringBuffer s2=new StringBuffer("Tenemos que decir cosas");
int posicion=s2.indexOf ("que");
System.out.println(posicion);
```

Resultado: 8

● **int indexOf (String str, int posicion):** devuelve la posición en la que aparece por primera vez la cadena que se le pasa como primer parámetro, comenzando la búsqueda a partir de la posición especificada en el segundo parámetro. Si la posición a partir de la que se inicia la búsqueda no existe o la cadena no aparece, devuelve –1.

Ejm:

```
StringBuffer frase=new StringBuffer("Tenemos que decir cosas");
int posicion=frase.indexOf ("que",3);
```

Resultado: posición: 8 (T e n e m o s q u e)

0 1 2 3 4 5 6 7 8

● **int lastIndexOf (String str):** devuelve la posición de la última aparición de la cadena que se pasa por parámetro, comenzando a buscar desde el principio de la misma.

Ejm:

```
StringBuffer s1= new StringBuffer("this is index of example");
int index1=s1.lastIndexOf("is");
System.out.println(index1);
```

Resultado: 5

● **int lastIndexOf (String str, int fin):** devuelve la posición de la última aparición de la cadena pasada como parámetro, comenzando a buscar desde el principio de la misma hasta la posición indicada en el segundo parámetro.

Ejm: StringBuffer frase=new StringBuffer("Tenemos que decir cosas");

int posicion=frase.lastIndexOf ("que", 3);

System.*out*.println(posicion);

Resultado: -1

Convertir un StringBuffer a String

- **String toString():** devuelve el resultado de convertir un StringBuffer a String.

  Ejm:

  StringBuffer s2=new StringBuffer("Ejemplo");

  String str=s2.toString();

  System.out.println(str); //String.out.println(s2.toString();

Métodos para concatenar y modificar cadenas

- **String concat (String s1):** Devuelve la cadena concatenando al final s1 (igual que poner +)

- **StringBuffer append (tipoDato a):** tipoDato puede ser un tipo básico, un String o un objeto. Convierte implícitamente a String si es necesario, el parámetro que recibe y lo anexa al StringBuffer.

  Son equivalentes las siguientes sentencias:

  Ejm:

  String s1="BC"+22;

  String s2=new StringBuffer().append ("BC").append(22).toString();

  System.out.println(s1+":"+s2);

  Resultado: BC22:BC22

  Si la adicción de caracteres hace que aumente el tamaño de StringBuffer más allá de su capacidad actual, el sistema asigna más memoria. Como la asignación de memoria es una operación relativamente cara, debemos hacer un código más eficiente inicializando la capacidad del StringBuffer de forma razonable para el primer

contenido, así minimizaremos el número de veces que se tendrá que asignar memoria.

- **StringBuffer insert (int posicion, tipoDato a) :** tipoDato puede ser un tipo básico, un String o un objeto. Convierte implícitamente a String si es necesario la variable tipoDato, y la inserta en la posición indicada por el primer parámetro que se le pasa al método.

  Ejm:

```
StringBuffer sb1 = new StringBuffer("01234567890");
sb1.insert (sb1.length(),98); // la longitud es: 11
sb1.insert (4,"ab");
System.out.println(sb1);  //Resultado: 0123ab456789098
```

- **StringBuffer reverse():** devuelve la cadena al revés.

  Ejm:

```
StringBuffer sb1 = new StringBuffer("Buenos días");
sb1.reverse();
System.out.println(sb1);
```

  Resultado: saíd soneuB

- **StringBuffer delete (int inicio, int fin):** elimina los caracteres desde inicio a (fin-1), incluidos ambos, del StringBuffer.

  Ejm:

```
StringBuffer sb1 = new StringBuffer("01234567890");
sb1.delete(1,3);
System.out.println(sb1);
```

  Resultado: 034567890

- **StringBuffer replace (int inicio, int fin, String str):** sustituye los caracteres desde inicio a (fin-1) que hay en el StringBuffer, por el String pasado por parámetro.

  Ejm:

```
StringBuffer sb1 = new StringBuffer("01234567890");
sb1.replace(6,8,"hola");
System.out.println(sb1);
```

  Resultado: 012345hola890

Extraer un substring de un StringBuffer

- **String substring (int i):** devuelve un String formado por los caracteres situados desde la posición que se pasa por parámetro (incluida), hasta el final de la cadena.

  Ejm:

  ```
  String st1;
  StringBuffer sb1 = new StringBuffer("01234567890");
  st1=sb1.substring(8);
  System.out.println(st1);
  ```

  Resultado: 890

- **String substring (int inicio, int fin):** Devuelve un String formado por los caracteres situados desde inicio hasta (fin -1), incluidos ambos, del StringBuffer. Si el número de caracteres de la cadena es menor que el indicado en el parámetro "inicio" se produce un error, al igual que si el parámetro "fin" sobrepasa la longitud del StringBuffer. Se puede evitar este tipo de errores si controlamos bien el número de caracteres de la cadena con la que trabajamos.

  Ejm:

  ```
  String st1;
  StringBuffer sb1 = new StringBuffer("01234567890");
  st1=sb1.substring(2,8);
  System.out.println(st1);
  ```

  Resultado: 234567