

MÉTODOS

Igual que el boletín anterior, se abordan conceptos muy importantes. Cuanto mejor entiendas dichos conceptos, con más seguridad podrás avanzar en el módulo.

Un objeto es una especie de cápsula formada por dos componentes: datos y métodos. Un sinónimo de método es *operación*. Los atributos guardan información sobre el objeto y los métodos, generalizando mucho, pueden hacer cosas con esa información.

Aprovechando que anteriormente estudiamos "funciones" en kotlin ¿Un método es lo mismo que una función?. Sin entrar en detalles tiquismiquis, podemos decir que un método no es más que una función asociada a un objeto. Más adelante iremos aprendiendo la relación y diferencias entre los conceptos función y método. Repasar las funciones Kotlin, te puede ayudar a entender bien este importante boletín.

Ejemplo de método.

Observa el siguiente ejemplo, en el que se hace evidente la conveniencia de que existan métodos.

```
class Coche {  
    String modelo;  
    int pasajeros;  
    int deposito;  
    int kpl;  
}  
  
class Unidad2 {  
    public static void main(String[] args) {  
        Coche peugeot308 = new Coche();  
        peugeot308.modelo = "Peugeot 308";  
        peugeot308.pasajeros = 5;  
        peugeot308.deposito = 60;  
        peugeot308.kpl = 20;  
        System.out.println("Modelo:" + peugeot308.modelo);  
        System.out.println("deposito:" + peugeot308.deposito);  
        System.out.println("kpl:" + peugeot308.kpl);  
        System.out.println("Autonomía:" + peugeot308.deposito * peugeot308.kpl);  
    }  
}
```

El cálculo de *autonomía* en el ejemplo anterior, podríamos verlo como poco elegante, ya que quizá la autonomía es una característica bastante "interior" de un objeto y el propio objeto debería ofrecer el cálculo. Esto se soluciona escribiendo un método en la definición de la clase Coche.

```
class Coche {  
    String modelo;  
    int pasajeros;  
    int deposito;  
    int kpl;
```

```

void autonomia() {
    System.out.println("Autonomía:" + deposito * kpl);
}

}

class Unidad2 {

    public static void main(String[] args) {
        Coche peugeot308 = new Coche();
        peugeot308.modelo = "Peugeot 308";
        peugeot308.pasajeros = 5;
        peugeot308.deposito = 60;
        peugeot308.kpl = 20;
        System.out.println("Modelo:" + peugeot308.modelo);
        System.out.println("deposito:" + peugeot308.deposito);
        System.out.println("kpl:" + peugeot308.kpl);
        peugeot308.autonomia();
    }
}

```

Valores calculados vs valores almacenados

Observa que, en el caso del ejemplo anterior, no sería apropiado definir un atributo *autonomía* ya que antes de usar su valor siempre habría que comprobar de alguna manera si cambió *kpl* o *depósito* para recalcular *autonomía*, con lo cual es inútil almacenar este valor, es mejor calcularlo.

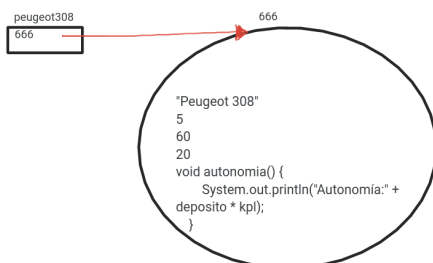
EN JAVA, AL CREAR UN OBJETO SE COPIA EN SU INTERIOR EL CÓDIGO DE LOS MÉTODOS

Parece una observación poco útil y de bajo nivel pero es importante conocerla para entender ciertos comportamientos.

Este comportamiento puede resultar curioso o "bruto" ya que por ejemplo, si el `main()` anterior hubiera creado 1000 coches cada uno de esos objetos coche almacena el código de `Autonomia()`. Recuerda esto un poco a que todas y cada una de las células de nuestro cuerpo almacena la misma copia de nuestro material genético.

Este comportamiento tuvo siempre buenos resultados, si bien es cierto que lenguajes más modernos como python ya no hacen este almacenamiento interno.

Del código anterior, que crea un objeto de tipo coche, que imaginamos que el `new` lo creo en la posición de memoria 666, podemos imaginar que la situación en memoria es la siguiente:



Sobre el nombre de los métodos: Mejor en infinitivo

Otra observación es que el nombre de los métodos debe indicar una acción y normalmente se expresan en infinitivo por lo que un nombre más apropiado

para el caso anterior sería por ejemplo `calcularAutonomia()`. Esto es lo más habitual, pero no es una regla sagrada.

Un método se define una vez, y se puede usar muchas.

El método es un mecanismo inspirado en el concepto de función matemática. En el mundo de la programación este concepto tiene varios objetivos y uno inmediato es que permite evitar escribir código duplicado. El código duplicado es una característica indeseable a la hora de escribir código ya que lo convierte en código de baja calidad. Observa el siguiente ejemplo

```
class Coche {
    String modelo;
    int pasajeros;
    int deposito;
    int kpl;

    void imprimirCoche() {
        System.out.println("Modelo:" + modelo);
        System.out.println("Nº de pasajeros: " + pasajeros);
        System.out.println("capacidad deposito: " + deposito);
        System.out.println("consumo(Kilometros por litro): " + kpl);
    }
}

class Unidad2 {
    public static void main(String[] args) {
        Coche peugeot308 = new Coche();
        peugeot308.modelo = "Peugeot 308";
        peugeot308.pasajeros = 5;
        peugeot308.deposito = 60;
        peugeot308.kpl = 20;
        peugeot308.imprimirCoche();
        peugeot308.imprimirCoche();
        peugeot308.imprimirCoche();
    }
}
```

Sin usar el mecanismo de "método", observa el código duplicado que surge en el `main()`

```
class Coche {
    String modelo;
    int pasajeros;
    int deposito;
    int kpl;
}

class Unidad2 {
    public static void main(String[] args) {
        Coche peugeot308 = new Coche();
        peugeot308.modelo = "Peugeot 308";
        peugeot308.pasajeros = 5;
        peugeot308.deposito = 60;
        peugeot308.kpl = 20;

        System.out.println("Modelo:" + peugeot308.modelo);
        System.out.println("Nº de pasajeros: " + peugeot308.pasajeros);
        System.out.println("capacidad deposito: " + peugeot308.deposito);
        System.out.println("consumo(Kilometros por litro): " + peugeot308.kpl);

        //otra impresión

        System.out.println("Modelo:" + peugeot308.modelo);
        System.out.println("Nº de pasajeros: " + peugeot308.pasajeros);
        System.out.println("capacidad deposito: " + peugeot308.deposito);
        System.out.println("consumo(Kilometros por litro): " + peugeot308.kpl);
    }
}
```

```
//y otra

System.out.println("Modelo:" + peugeot308.modelo);
System.out.println("Nº de pasajeros: " + peugeot308.pasajeros);
System.out.println("capacidad deposito: " + peugeot308.deposito);
System.out.println("consumo(Kilometros por litro): " + peugeot308.kpl);

}
}
```

El código duplicado genera un código de mala calidad que es complicado de mantener, supongamos por ejemplo la siguiente situación: Escribo el código anterior y pasado un tiempo observo que me olvidé de algunos acentos (depósito y kilómetros) y que no empecé cada línea en mayúscula. Arreglar esto en la versión con método es sencillo, se corrige cada error una sola vez, en la definición del método. La versión sin método hay que revisar todo el código para ver donde imprimo las características del coche e ir corrigiendo por todos lados.

LOS MÉTODOS DEVUELVEN VALORES CON RETURN

Los métodos devuelven un valor al acabar su ejecución. Es obligatorio en su definición indicar el tipo de valor que devuelve un método (int, double, ...). Si no devuelven ningún valor se indica con la palabra reservada *void*

En el ejemplo anterior el método autonomía no devuelve ningún valor y por eso se indica que es de tipo void que significa "vacío" es decir, que no devuelve ningún valor

Podemos modificar la definición de clase anterior haciendo que el método calcularAutonomia() simplemente haga el cálculo, y que el main() se encargue de imprimir el mensaje deseado.

```
int calcularAutonomia(){
    return deposito*kpl;
}
```

con la definición anterior indicamos:

- que el método devuelve un entero
- el valor(entero) que quiero que devuelva se indica con return. Observa que si multiplico un entero por otro entero el resultado es otro entero

Ahora en main() podría hacer:

```
System.out.println("La autonomía es: " + peugeot308.calcularAutonomia());
o si lo prefieres
int a=peugeot308.calcularAutonomia();
System.out.println("La autonomía es: " + a);
```

Ejercicio U2_B3_E1: Cambia el siguiente código

```
class Coche {
    String modelo;
    int pasajeros;
    int deposito;
    int kpl;

    void calcularAutonomia() {
        System.out.println("Autonomía: " + deposito * kpl);
    }
}
```

```

    }
}

class Unidad2 {

    public static void main(String[] args) {
        Coche peugeot308 = new Coche();
        peugeot308.modelo = "Peugeot 308";
        peugeot308.pasajeros = 5;
        peugeot308.deposito = 60;
        peugeot308.kpl = 20;
        System.out.println("Modelo:" + peugeot308.modelo);
        System.out.println("deposito:" + peugeot308.deposito);
        System.out.println("kpl:" + peugeot308.kpl);
        peugeot308.autonomia();
    }
}

```

De forma que calcularAutonomia() devuelva un int como se explicó anteriormente.

Sobre el tipo de la expresión del return y el tipo del método.

La norma genérica es: Tienen que ser del mismo o tipo habrá error de compilación.

Hay algunas excepciones:

- Si los tipos son primitivos y la expresión del return se puede promocionar por upcasting al tipo del método.
- Si hay relación de herencia entre los tipos (ya lo estudiaremos más adelante)

Return y métodos que devuelven void

- Un método que no devuelve void necesariamente tiene que contener en su cuerpo al menos un return
- Si el método devuelve void, se puede no escribir ningún return y se asume un return al final del método. Puede interesar, como veremos más adelante incluir también return en un método void, y en este caso no se escribe

return void
 Si no simplemente
return

Ejemplo: usamos explícitamente return en el siguiente ejemplo de método void

```

void autonomia() {
    System.out.println("Autonomía:" + deposito * kpl);
    return; // No necesario
    //recuerda que no se puede poner return void
}

```

MÉTODOS CON PARÁMETROS

Indicamos que el concepto de método se basa en el concepto de función matemática. Las funciones matemáticas igual que los métodos primero se definen y luego se usan.

Por ejemplo, definir una función f que dado un x me devuelve su cuadrado

$$f(x)=x^2$$

Ahora puedo usar la función. A "usar" también se le llama "invocar" o "llamar" a la función para que haga el cálculo correspondiente.

Por ejemplo $f(2)$ que devuelve el valor 4, $f(3)$ que devuelve el valor 9 etc.

A la variable x de la definición de la función se le llama parámetro y a los valores concretos que toma x se les llama argumentos.

Respecto a los métodos ya vimos que al igual que una función:

- Se definen: es decir se escriben en la clase
- Se invocan: utilizando el operador punto sobre una variable referencia.

Y además ahora al igual que con las funciones:

- Al definir un método puedo incluir parámetros
- Al invocar una función puedo pasarle un valor (argumento) a dicho parámetro.

Los parámetros no son más que variables locales que se definen entre los paréntesis del método y que se inicializan cada vez que se llama al método con el valor del argumento.

Ejemplo: Añadimos a *Coche*, un método *gasofaNecesaria()* con un parámetro "kilometros" que va a contener los kilómetros que se quieren recorrer y nos devuelve el consumo que se va a realizar

```
double gasofaNecesaria(int kilometros){
    return (double) kilometros/kpl;
}
```

para invocar al método incorporaríamos en *main()* una instrucción que incorpore la llamada:

```
peugeot308.gasofaNecesaria(100);
```

donde 100, es el valor o argumento que va a recibir el parámetro kilómetros.

```
class Coche{
    int pasajeros; //número de pasajeros
    int deposito; //capacidad del depositos en litros
    int kpl; //kilometros que se pueden recorrer con un litro,
    int calcularAutonomia(){
        return deposito*kpl;
    }

    double gasofaNecesaria(int kilometros){
        return (double) kilometros/kpl;
    }
}

class Unidad2 {

    public static void main(String[] args) {
        Coche peugeot308 = new Coche();
        peugeot308.pasajeros=5;
        peugeot308.deposito=60;
        peugeot308.kpl=20;

        System.out.print("La autonomía de peugeot 308 es " + peugeot308.calcularAutonomia());
        System.out.println(" . Y para recorrer 100 kilometros se necesitan " + peugeot308.gasofaNecesaria(100) + "
litros");

        Coche mercedesXZ= new Coche();
        mercedesXZ.pasajeros=7;
        mercedesXZ.deposito=100;
        mercedesXZ.kpl=10;
        System.out.print("La autonomía de mercedes XZ es " + mercedesXZ.calcularAutonomia());
```

```

        System.out.println(" . Y para recorrer 100 kilometros se necesitan " + mercedesXZ.gasofaNecesaria(100) + "
        litros");
    }
}

```

Mini explicación del mecanismo de parámetros.

Tomando como ejemplo el caso anterior.

La instrucción `peugeot308.gasofaNecesaria(100)` hace que para el objeto referenciado por `peugeot308` se ejecute el método `gasofaNecesaria()` de la siguiente forma:








1. Un parámetro no es más que una variable local y un argumento no es más que una expresión que se evalúa y de dicha evaluación se obtiene un valor. Entre el parámetro y el argumento, se provoca internamente antes de la ejecución de las instrucciones del método, una "suerte" de sentencia de asignación. Para la instrucción anterior ocurre pues de forma imaginaria :
`Kilometros=100`
2. Se ejecuta el resto del método con el valor 100 para la variable `kilometros`.

Analizar con el tracer de Bluej las invocaciones a métodos

Ejecuta el ejemplo anterior con Bluej y utiliza "step into" ("entrar en") de la ejecución paso a paso.

Ejercicio U2_B3_E2:

Escribe la siguiente clase `Bicicleta`

 Bicicleta
 ~int velocidad  ~int marcha
 ~void cambiarMarcha(int novoValor)  ~void acelerar(int incremento)  ~void frear(int decremento)  ~void imprimirEstado()

De forma que sea manejada por el siguiente main

```
class Unidad2{
    public static void main(String[] args) {
        // Crea dos obxectos bicicleta
        Bicicleta bicicleta1 = new Bicicleta();
        Bicicleta bicicleta2 = new Bicicleta();
        // Invoca os métodos destes obxectos
        bicicleta1.acelerar(10);
        bicicleta1.cambiarMarcha(2);
        bicicleta1.imprimirEstado();
        bicicleta2.acelerar(10);
        bicicleta2.cambiarMarcha(2);
        bicicleta2.acelerar(10);
        bicicleta2.cambiarMarcha(3);
        bicicleta2.imprimirEstado();
    }
}
```

SETTER Y GETTER: MÉTODOS PARA ACCEDER A LOS ATRIBUTOS DE UN OBJETO.

Por cuestiones de diseño OO, se debe evitar el acceso directo a los atributos de un objeto desde el exterior, es decir, desde otros objetos. El acceso a los atributos de un objeto debe de ser siempre a través de métodos de dicho objeto que pueden hacer comprobaciones de errores y otras cuestiones. Si evitamos el acceso directo a los atributos nos encontramos con la necesidad de dos tipos de métodos:

- métodos *setter* o *colocadores* para especificar el valor de un atributo.
- métodos *getter* u *obtenedores* para obtener el valor de un atributo.

Cada atributo debería tener su setter y getter correspondiente.

Ejemplo: incorporamos métodos de acceso a los atributos de la clase Coche

```
class Coche {

    String modelo;
    int pasajeros;
    int deposito;
    int kpl;

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String m) {
        modelo =m;
    }

    public int getPasajeros() {
        return pasajeros;
    }
}
```



```

    public void setPasajeros(int p) {
        pasajeros = p;
    }

    public int getDeposito() {
        return deposito;
    }

    public void setDeposito(int d) {
        deposito = d;
    }

    public int getKpl() {
        return kpl;
    }

    public void setKpl(int k) {
        kpl = k;
    }

    void calcularAutonomia() {
        System.out.println("Autonomía:" + deposito * kpl);
    }

    double gasofaNecesaria(int kilometros) {
        return (double) kilometros / kpl;
    }
}

class Unidad2{
    public static void main(String[] args) {
        Coche citroenC1= new Coche();
        citroenC1.setModelo("Citroen C1 special");
        citroenC1.setPasajeros(4);
        citroenC1.setDeposito(50);
        citroenC1.setKpl(25);
        //citroenC1.kpl=25;

        System.out.println("un citroen C1 permite " + citroenC1.getPasajeros() + " pasajeros");
        System.out.println("un citroen C1 tiene consumo de " + citroenC1.getKpl() + " kilómetros por
litro");
    }
}

```

Observa el "estilo" de los nombres de métodos setter y getter. Son de la forma `setAtributo()` y `getAtributo()`. También se pueden ver más raramente en castellano como `colocarAtributo()` y `obtenerAtributo()`. Mejor en inglés que es como el 99% de los programadores java lo hacen. Es una cuestión de estilo, no sintáctica, que mejora la lectura de los programas. Si en lugar de `getPasajero()` utilizara el nombre `paraPasajeros()` la intención del método no la comprendo hasta analizar su código pero si veo `"getPasajero()"` sé al instante que ese método me devuelve el valor de un atributo que se llama pasajero. Si hace otra cosa, es que el programador que lo escribió no sigue las normas de estilo Java y se llevará mal con la comunidad Java.

Ejemplo : puedes acceder al API y comprobar que casi todas las clases tiene este tipo de métodos. Por ejemplo con las clases `GregorianCalendar` y `JFrame` para observar que tienen muchos métodos `set/get`

Atributos privados

¿Te resulta curioso que se obtenga el mismo resultado con estas dos instrucciones?

```
citroenC1.setKpl(25);
```

```
citroenC1.kpl=25;
```

A la vista de las instrucciones anteriores y sabiendo que el método es

```
void setKpl(int k){  
    kpl=k;  
}
```

Nos preguntamos ¿Para qué valen los métodos `set` y `get`? ¿No son superfluos?. Es una cuestión compleja que iremos viendo poco a poco, por el momento y generalizando mucho debemos observar que, *cuando los atributos son privados, a los objetos de otras clases no les queda más remedio que acceder a los atributos con `setter/getter`.*

Más adelante, estudiaremos en profundidad los diferentes niveles de acceso a un atributo. Por el momento hay que saber que:

- Hasta ahora, estuvimos usando el nivel de acceso por defecto que permite acceder libremente a los atributos de las clases que están dentro del mismo paquete .
- Hay otro nivel de acceso llamado *private*. Los atributos *private* no son accesibles desde otras clases.

Lógicamente, al contrario que los atributos los métodos `set/get` no tiene sentido que sean privados.

Ejemplo:

hacemos *private kpl* y por tanto deja de estar accesible para el código que no es de la clase `Coche`:

Indicamos que el atributo `kpl` es *private*

```
class Coche {  
  
    String modelo;  
    int pasajeros;  
    int deposito;  
    private int kpl;  
    //resto de clase igual que antes  
  
}  
  
class Unidad2 {  
  
    public static void main(String[] args) {  
        Coche citroenC1= new Coche();  
        citroenC1.kpl=25;    //ERROR
```

```
}  
}
```

```
L:\Programacion>javac Unidad2.java  
Unidad2.java:54: error: kpl has private access in Coche  
    citroenC1.kpl=25;  
                ^
```

La clase Unidad2 no puede acceder a los miembros privados de Coche

Ejemplo: Si usamos setKpl() no hay problema ya que setKpl() no es private y es accesible por Unidad2, luego como setKpl() pertenece a la clase Coche puede acceder sin problemas al atributo kpl aunque sea privado.

```
class Unidad2{  
    public static void main(String[] args) {  
        Coche citroenC1= new Coche();  
        //citroenC1.kpl=25;  
        citroenC1.setKpl(25);  
    }  
}
```

Al forzar a Unidad2 a que use setKpl para modificar el atributo kpl podemos imponer un control sobre la modificación, por ejemplo obligamos a que los valores sean positivos.

```
public void setKpl(int k) {  
    if(k>0){  
        kpl=k;  
    }  
    //si k no es positivo no hace nada  
}
```

Comprueba con

```
class Unidad2{  
    public static void main(String[] args) {  
        Coche citroenC1= new Coche();  
  
        citroenC1.setKpl(-2);  
        //citroenC1.setKpl(2);  
        System.out.println(citroenC1.getKpl());  
    }  
}
```

Ejercicio U2_B3_E3:

Añade a la clase Persona los get/set necesarios para que funcione el main de Unidad2

```
class Persona{  
    private String nombre;  
    private int edad;  
    //añadir métodos  
}
```

```
class Unidad2{
```

```

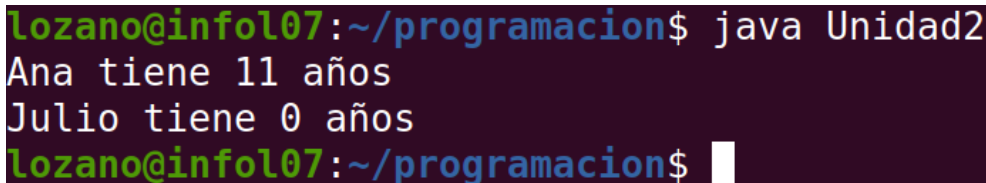
public static void main(String[] args) {
    Persona persona1= new Persona();
    Persona persona2= new Persona();
    //persona1.nombre="Ana" //ierror!
    persona1.setNombre("Ana");
    persona1.setEdad(11);

    persona2.setNombre("Julio");
    persona2.setEdad(-16);

    System.out.println(persona1.getNombre()+" tiene "+ persona1.getEdad()+" años");
    System.out.println(persona2.getNombre()+" tiene "+ persona2.getEdad()+" años");

}
}

```



```

lozano@infol07:~/programacion$ java Unidad2
Ana tiene 11 años
Julio tiene 0 años
lozano@infol07:~/programacion$

```

Observa que el método `setEdad()` corrige las edades negativas. Si recibe un valor negativo asigna como edad 0

FIN DE EJECUCIÓN DE UN MÉTODO

Un método acaba su ejecución, es decir, devuelve el control al llamante:

1. si llegamos al final del método, es decir, llegamos a la llave de cierre del método " `}`". Esto sólo es posible en el caso de los métodos que devuelven `void`.
2. Si nos encontramos con un *return*. Puede haber varios *return* en un método, más adelante veremos más sobre esto.

Comprueba esto en los ejemplos que vimos en este boletín

CONSTRUCTORES

Un constructor es un método especial que se ocupa de inicializar un objeto cuando se crea con el operador `new`. Si no definimos el constructor/-es de una clase, java le asigna uno por defecto.

Constructor por defecto

Si no definimos un constructor para una clase dada, el operador *new* utiliza el constructor por defecto que se llama exactamente igual que la clase seguido de paréntesis vacíos, es decir, sin parámetros

Ejemplo: Ejecuta el siguiente código

```

class MiClase{
    int x;
}

class Unidad2{
    public static void main(String args[]){
        MiClase obj1 = new MiClase(); //estamos utilizando el constructor por defecto
        System.out.println("obj1.x: " + obj1.x); //observa como el constructor inicializa x.
    }
}

```

Observa que el constructor por defecto inicializa los atributos. En este caso, sólo tenemos un atributo y es de tipo numérico por lo que lo inicializa a 0.

Escribir un constructor explícito

Un constructor tiene dos características importantes:

- nunca devuelve un valor, ni siquiera *void*.
- se llama exactamente igual que su clase seguido de los paréntesis de método.

Clase: MiClase

Constructor: MiClase()

```
class MiClase{
    int x;
    void MiClase(){ //observa como el constructor no devuelve ningún valor
        System.out.println("inicializando x ....");
        x=10;
    }
}

class Unidad2{
    public static void main(String args[]){
        MiClase obj1;
        obj1 = new MiClase(); //new invoca el constructor
        MiClase obj2 = new MiClase();
        System.out.println("obj1.x: " + obj1.x);
        System.out.println("obj2.x: " + obj2.x);
    }
}
```

Ahora podemos entender un poco mejor el operador new:

1. Reserva memoria para el objeto
2. Devuelve la dirección donde comienza a almacenarse el objeto.
3. Invoca la ejecución del constructor de la clase correspondiente.

Después de lo visto: Intenta explicar porqué el siguiente código genera valores 0 para X.

```
class MiClase{
    int x;
    Void MiClase(){ //observa como el constructor no devuelve ningún valor
        System.out.println("inicializando x ....");
        x=10;
    }
}

class Unidad2{
    public static void main(String args[]){
        MiClase obj1;
        obj1 = new MiClase(); //new invoca el constructor
        MiClase obj2 = new MiClase();
        System.out.println("obj1.x: " + obj1.x);
        System.out.println("obj2.x: " + obj2.x);
    }
}
```

Constructores con parámetros

Mismo efecto que en métodos normales

Ejemplo: Ejecuta el siguiente código

```

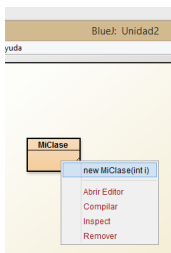
class MiClase{
    int x;
    MiClase(int i){
        x=i;
    }
}

class Unidad2{
    public static void main(String args[]){
        MiClase obj1 = new MiClase(4);
        MiClase obj2 = new MiClase(25);
        System.out.println("obj1.x: " + obj1.x);
        System.out.println("obj2.x: " + obj2.x);
    }
}

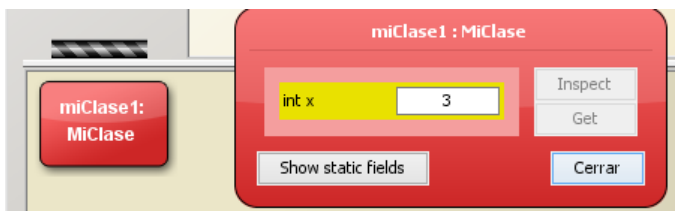
```

Ejercicio: Ahora que tenemos constructor con parámetros, probamos la creación de objetos desde bluej sin necesidad de crearlos desde otra clase.

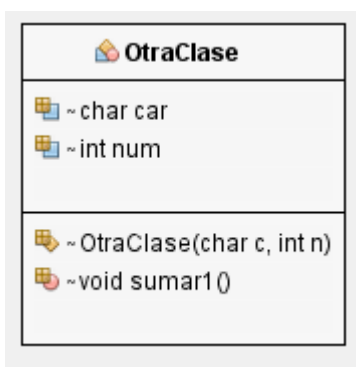
Invocamos al constructor, le damos un valor a x, por ejemplo 3



Observo que se creó una instancia(un objeto) que bluej para identificarlo llama miClase1 (se puede cambiar este nombre) y si hago 2 clic sobre él puedo ver su estado



Ejercicio U2_B3_E4: Crea *OtraClase*

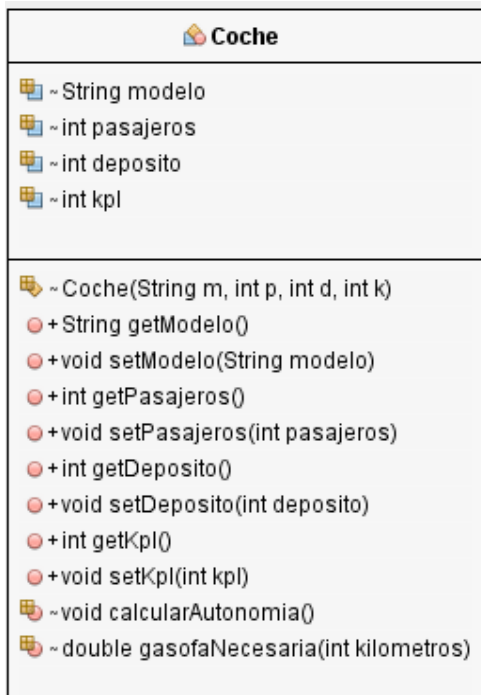


sumar1() incrementa en 1 el atributo num

Prueba el constructor desde el main de **Unidad2** y crea también objetos de OtraClase directamente desde BlueJ

Ejercicio U2_B3_E5: Un constructor para la clase Coche.

Hasta ahora utilizamos para crear objetos de la clase Coche el constructor por defecto. Modifica el ejemplo de la clase coche(la versión con setters y getters) añadiendo el constructor



- Y desde main() probamos el constructor

```
Coche coche1 = new Coche("Citroen C1", 5, 55, 18);
System.out.println("coche1:");
System.out.println("\tmodelo: " + coche1.getModelo());
System.out.println("\tpasajeros: " + coche1.getPasajeros());
System.out.println("\tdeposito: " + coche1.getDeposito());
System.out.println("\tkpl: " + coche1.getKpl());
```

- Como Coche es una clase con una variedad de métodos, práctica a crear directamente con BlueJ objetos Coche y ejecuta sobre ellos algún método.

Constructor por defecto en clases con algún constructor definido.

Una vez que una clase tiene al menos un constructor definido ya no se puede usar el constructor por defecto

Ejemplo: con el código del anterior ejercicio, en el que la clase Coche tiene un constructor definido comprueba que ocurriría si añadimos la siguiente instrucción:

```
Coche coche2 = new Coche();
```

Compila y observa que: ¡ya no se puede usar el constructor por defecto!

Sobre esto volveremos a hablar más adelante, por el momento es suficiente darse cuenta del error

Ejercicio U2_B3_E6: Escribe un constructor para la clase Persona

```
class Persona{
    String nombre;
    int edad;
    //falta constructor
}

class Unidad2{
    public static void main(String[] args) {
        Persona persona= new Persona("Telma",11);
        System.out.println(persona.nombre+" tiene "+ persona.edad+" años");
    }
}
```

Inicializar un atributo en su definición o en un constructor

¿Cuanto vale x?

```
class A{
    int x=2;
    A(){
        x=3;
    }
}

class Unidad2{
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x);
    }
}
```

Se usan las dos cosas, pero no a la vez como en el ejemplo. Se prefiere un método u otro según el contexto. En programas grandes se suele hacer siempre en constructores. En el ejemplo anterior, como se inicializa x en dos puntos podemos observar que finalmente prevalece la inicialización del constructor. Comprueba esto ejecutando el código anterior.

MÁS DE MÉTODOS: USAR UNA VARIABLE REFERENCIA COMO PARÁMETRO.

Esto es un poco duro por el momento ya que nos falta práctica pero empezamos a meterle diente ...

Ya vimos lo que es un parámetro. Observa que en los ejemplos los parámetros eran siempre variables de tipos primitivos. También se pueden emplear como parámetros variables referencia. Si un parámetro es una referencia, le estamos pasando al

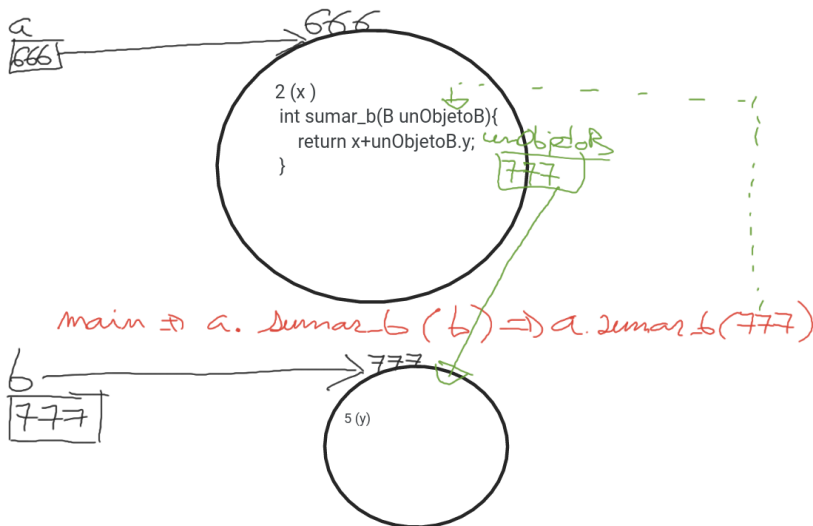
método la referencia(dirección) de un objeto y por tanto tiene acceso a dicho objeto.

Ejemplo sumar_b() tiene como parámetro una referencia de tipo B

```
class B{
    int y =5;
}
class A{
    int x=2;
    int sumar_b(B unObjetoB){
        return x+unObjetoB.y;
    }
}

class Unidad2{
    public static void main(String[] args) {
        A a = new A();
        B b=new B();

        int suma =a.sumar_b(b);
        System.out.println(suma);
    }
}
```



Podemos demostrar que efectivamente la variable unObjetoB permite acceder al mismo objeto que la variable b ya que ambas contienen el mismo valor de referencia imprimiendo los valores de las referencias y comprobando que son iguales

```
class Unidad2{
    public static void main(String[] args) {
        A a = new A();
        B b=new B();
        System.out.println("referencia que contiene a: "+a);
        System.out.println("referencia que contiene b: "+b);
        int suma =a.sumar_b(b);
        System.out.println(suma);
    }
}

class B{
    int y =5;
```

```

}
class A{
    int x=2;
    int sumar_b(B unObjetoB){
        System.out.println("referencia que contiene unObjetoB: "+unObjetoB);
        return x+unObjetoB.y;
    }
}

```

Ejemplo: averiguar si un coche tiene más autonomía que otro.

primero lo resolvemos con el calcularAutonomia() sin parámetros

```

class Coche {

    String modelo;
    int pasajeros;
    int deposito;
    int kpl;

    Coche(String m, int p, int d, int k) {
        modelo = m;
        pasajeros = p;
        deposito = d;
        kpl = k;
    }

    int calcularAutonomia() {
        return deposito * kpl;
    }

}

class Unidad2 {

    public static void main(String[] args) {
        Coche coche1 = new Coche("un coche", 5, 60, 20);
        Coche coche2 = new Coche("otro coche", 7, 70, 30);
        System.out.println("¿Tiene coche1 más autonomía que coche2? " + (coche1.calcularAutonomia() >
coche2.calcularAutonomia()));
    }

}

```

Vamos a conseguir lo mismo pero usando un método al que se le pasa como parámetro una variable referencia tipo coche.

```

boolean mayorAutonomia(Coche c){
    return calcularAutonomia()>c.calcularAutonomia();
}

```

El ejemplo completo:

```

class Coche {

    String modelo;
    int pasajeros;
    int deposito;
    int kpl;

```

```

Coche(String m, int p, int d, int k) {
    modelo = m;
    pasajeros = p;
    deposito = d;
    kpl = k;
}

int calcularAutonomia() {
    return deposito * kpl;
}

boolean mayorAutonomia(Coche c) {
    return calcularAutonomia() > c.calcularAutonomia();
    //return deposito*kpl >= c.deposito*c.kpl;
}

}

class Unidad2 {

    public static void main(String[] args) {
        Coche coche1 = new Coche("un coche", 5, 60, 20);
        Coche coche2 = new Coche("otro coche", 7, 70, 30);
        System.out.println("¿Tiene coche1 más autonomía que coche2? " + coche1.mayorAutonomia(coche2));
    }
}

```

Podemos imaginarnos que en la llamada al método `coche1.mayorAutonomia(coche2)` ocurre lo siguiente:

1. Se copia el valor de `coche2` en `c`, es decir `c=coche2`;
2. Ya con valores concretos para la variable local `c`, se ejecuta el método para el objeto referenciado por `coche1`, ya que es sobre el objeto que se invoca el método.

Este tipo de código se entiende mejor utilizando *this*. *this* lo estudiamos en un boletín posterior.

Otro ejemplo: Más difícil todavía. Ahora `mayorAutonomia()` en lugar `true` o `false`, devuelve un objeto `coche`, aquel que tiene más autonomía de los dos.

```

class Coche {

    String modelo;
    int pasajeros;
    int deposito;
    int kpl;

    Coche(String m, int p, int d, int k) {
        modelo = m;
        pasajeros = p;
        deposito = d;
        kpl = k;
    }

    int calcularAutonomia() {

```

```

        return deposito * kpl;
    }

    Coche mayorAutonomia(Coche c){
        Coche cocheMayorAut=new Coche("nada",0,0,0);
        if(calcularAutonomia()>c.calcularAutonomia()){
            cocheMayorAut.modelo=modelo;
            cocheMayorAut.pasajeros=pasajeros;
            cocheMayorAut.deposito=deposito;
            cocheMayorAut.kpl=kpl;
        }else{
            cocheMayorAut.modelo=c.modelo;
            cocheMayorAut.pasajeros=c.pasajeros;
            cocheMayorAut.deposito=c.deposito;
            cocheMayorAut.kpl=c.kpl;
        }
        return cocheMayorAut;
    }
}

```

Quizá te resulte difícil de entender en estos momentos, intenta captar la mayor cantidad de detalles posibles. Clava tu mirada en el return y en el tipo del método ya que es lo más novedoso.

Ejercicio U2_B3_E7: Hacer un main en Unidad2 que pruebe el nuevo método mayorAutonomia()

REFERENCIA NULL Y Nullpointerexception

Una referencia es una variable que guarda la dirección de un objeto. Para indicar que en un momento dado una variable referencia no referencia a ningún objeto se le da el valor especial *null*.

Si una referencia vale *null* no tiene sentido utilizar con ella el operador "." para acceder a los miembros de un objeto ya que si no hay objeto, no hay miembros. Igual que cuando dividimos por cero (división entera) saltaba una excepción "Division by zero", si con una referencia null intentamos acceder a un miembro salta una excepción "Nullpointerexception". En el siguiente ejemplo mc.x es realmente null.x y como se considera un error de programación java para la ejecución del programa.

Comprueba que mc.x genera una "null pointer Exception".

```

class MiClase{
    int x=2;
}

class Unidad2{
    public static void main(String[] args) {
        MiClase mc=null;
        System.out.println("mc contiene el valor: "+mc);
        System.out.println("quiero ver que contiene el miembro x: "+mc.x);
        System.out.println("hola");
    }
}

```

Igual que un programador tiene la responsabilidad de que en su código nunca ocurra una división por cero, también tiene la responsabilidad de evitar usar referencias nulas con el operador “.”

REFERENCIA NULL EN MÉTODOS

Cuando un programador escribe un método con algún parámetro, tiene que prever que otros programadores (o incluso el mismo) pueden mandar a ese método argumentos con valor null. Así que normalmente, para evitar que se genere un *null pointer exception* en el método se comprueba si la referencia es null y se procura, si tiene sentido, evitar la null pointer exception. Por ejemplo en el primer ejemplo visto de método con parámetro referencia, podemos decidir que si llega una referencia null entonces simplemente devolvemos el valor de x. Si no encontráramos un sentido a qué hacer cuando nos llega una referencia null pues no habría más remedio que dejar que una null pointer exception ocurra.

Recuerda el ejemplo visto:

```
class B{
    int y =5;
}
class A{
    int x=2;
    int sumar_b(B unObjetoB){
        return x+unObjetoB.y;
    }
}
```

En un programa profesional el método sumar_b debería tener en cuenta que hacer si ocurriera que unObjetoB llega con valor null

```
class B{
    int y =5;
}
class A{
    int x=2;
    int sumar_b(B unObjetoB){
        if(unObjetoB==null){//si vale null simplemente devolvemos x
            return x;
        }else{
            return x+unObjetoB.y;
        }
    }
}

class Unidad2{
    public static void main(String[] args) {
        A obj_a = new A();
        B obj_b=null;

        int suma =obj_a.sumar_b(obj_b);
        System.out.println(suma);
    }
}
```

MÉTODOS STATIC

El concepto de `static` lo estudiaremos más adelante con más rigor, por el momento basta saber que puedo escribir un método que no pertenece a ningún objeto en particular sino a una clase. No es necesario crear un objeto para invocar a un método `static`.

Observa en el ejemplo las sintaxis:

- Invocar a un método no `static`:
`variablereferencia.NombreDemetodo`
- Invocar a un método `static`:
`NombreDeClase.NombreDeMétodoStatic`

```
class MiClase{
    static void miMetodoStatic(){
        System.out.println("mi método static");
    }
    void miMetodoNOStatic(){
        System.out.println("mi método NO static");
    }
}

class Unidad2{

    public static void main(String args[]) {
        MiClase.miMetodoStatic();
        MiClase mc=new MiClase();
        mc.miMetodoNOStatic();
    }
}
```

Hay muchas clases de la librería java que contienen métodos `static`, veamos algunos ejemplos.

SOBRE EL MÉTODO MAIN

En estos momentos del intrigante `main()` sabemos lo siguiente:

- Es un método los paréntesis `main()` lo delatan!
- Devuelve el valor `void`
- Tiene un parámetro pero que de momento no entendemos `iString[] args!`
- Es `static`. Cuando estudiemos `static` en profundidad explicaremos algo más, por el momento observa que como `main()` es `static` se puede ejecutar sin tener que crear un objeto `Unidad2` (no tenemos que hacer obligatoriamente `new Unidad2()`)

EJEMPLOS DE MÉTODOS STATIC DE LA CLASE MATH

La clase `Math`, como su nombre indica, tiene una serie de métodos que típicamente se necesitan para hacer cálculos matemáticos

```
class Unidad2{
    public static void main(String args[]){
        System.out.println(Math.sqrt(16));
        System.out.println(Math.pow(2,4));
        System.out.println(Math.max(34,56));
        System.out.println(Math.abs(-45));
        //etc.
    }
}
```

EJEMPLOS DE MÉTODOS STATIC DE LA CLASE INTEGER Y LONG

Aprovechamos este ejemplo para ilustrar que igual que hay métodos static, puede haber atributos static.

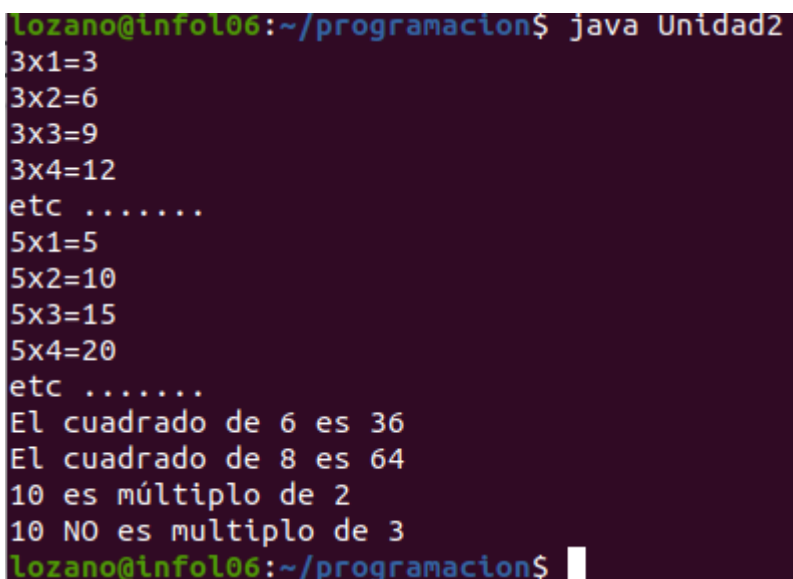
```
class Unidad2{
    public static void main(String args[]){
        System.out.println(Integer.parseInt("10"));
        //System.out.println(Integer.parseInt("diez")); //imal!
        System.out.println(Integer.toString(10));
        System.out.println(Integer.toBinaryString(10));
        //etc.
        System.out.println(Long.parseLong("10"));
        System.out.println(Long.toBinaryString(10));
        //etc.
        //también hay atributos static
        System.out.println(Integer.MAX_VALUE);
        System.out.println(Integer.MIN_VALUE);
        System.out.println(Long.MAX_VALUE);
        System.out.println(Long.MIN_VALUE);
    }
}
```

EJEMPLO DE MÉTODO STATIC PARA QUE EL PROGRAMA DETENGA SU EJECUCIÓN CIERTO TIEMPO.

El método sleep() de la clase Thread, detiene la ejecución del programa por el tiempo que se le indique en milisegundos. Para poder utilizar este método tenemos que añadir al main "otra cosa rara más" throws InterruptedException que más adelante entenderemos.

```
class Unidad2{
    public static void main(String args[]) throws InterruptedException{
        System.out.println("espera 3 segundos ...");
        Thread.sleep(3000);
        System.out.println("Hecho");
    }
}
```

Ejercicio U2_B3_E8: Crea la clase UtilidadesMate para que responda al siguiente main



```
lozano@info106:~/programacion$ java Unidad2
3x1=3
3x2=6
3x3=9
3x4=12
etc .....
5x1=5
5x2=10
5x3=15
5x4=20
etc .....
El cuadrado de 6 es 36
El cuadrado de 8 es 64
10 es múltiplo de 2
10 NO es multiplo de 3
lozano@info106:~/programacion$
```

```

class Unidad2 {
    public static void main(String[] args) {
        UtilidadesMate.imprimirTablaMultiplicar(3);
        UtilidadesMate.imprimirTablaMultiplicar(5);
        System.out.println("El cuadrado de 6 es "+UtilidadesMate.calcularCuadrado(6));
        System.out.println("El cuadrado de 8 es "+UtilidadesMate.calcularCuadrado(8));
        System.out.println(UtilidadesMate.aEsMultiploDeb(10,2));
        System.out.println(UtilidadesMate.aEsMultiploDeb(10,3));
    }
}

```

EL MÉTODO toString()

Sabemos que al incluir un variable referencia en un println() se imprime justamente el valor de dicha variable.

```

class Coche{
    String modelo;
    int numeroPasajeros;
}
public class Unidad2{
    public static void main(String[] args) {
        Coche c= new Coche();
        c.modelo="TESLA FIRE";
        c.numeroPasajeros=2;
        System.out.println(c);
    }
}

```

```

PS C:\Users\donlo\programacion> java Unidad2
Coche@4617c264
PS C:\Users\donlo\programacion>

```

A menudo cuando incorporamos una variable referencia en un println() lo que realmente nos gustaría imprimir es una representación en texto del estado del objeto, para ello podemos escribir un método, por ejemplo:

```

class Coche{
    String modelo;
    int numeroPasajeros;
    String convertirAString(){
        return "modelo: "+ modelo+ ", pasajeros: "+numeroPasajeros;
    }
}
public class Unidad2{
    public static void main(String[] args) {
        Coche c= new Coche();
        c.modelo="TESLA FIRE";
        c.numeroPasajeros=2;
        System.out.println(c.convertirAString());
    }
}

```

```

PS C:\Users\donlo\programacion> java Unidad2
modelo: TESLA FIRE, pasajeros: 2
PS C:\Users\donlo\programacion>

```

Pero la forma standard (lo que usa todo el mundo) es escribir en la clase un método toString(). Por razones que se escapan ahora en este boletín y entenderemos más adelante, este método hay que declararlo como public.

```

class Coche{
    String modelo;
    int numeroPasajeros;
    public String toString(){

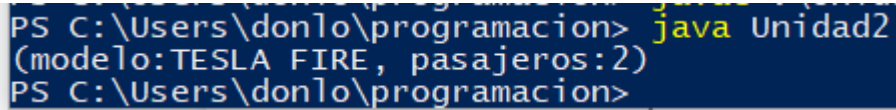
```



```

        return "(modelo:" + modelo + ", pasajeros:" + numeroPasajeros + ")";
    }
}
public class Unidad2{
    public static void main(String[] args) {
        Coche c= new Coche();
        c.modelo="TESLA FIRE";
        c.numeroPasajeros=2;
        System.out.println(c.toString());
    }
}

```



```

PS C:\Users\donlo\programacion> java Unidad2
(modelo:TESLA FIRE, pasajeros:2)
PS C:\Users\donlo\programacion>

```

Puedes comprobar que si retiramos el public en toString() se genera error de compilación.

VENTAJAS DE toString()

- Es standard, todo el mundo lo usa
- En el println automáticamente escribiendo simplemente el nombre de la referencia se invoca el toString().

Así el main anterior lo pudimos escribir (observa println())

```

public class Unidad2{
    public static void main(String[] args) {
        Coche c= new Coche();
        c.modelo="TESLA FIRE";
        c.numeroPasajeros=2;
        System.out.println(c);
    }
}

```