

1º CFGS DESARROLLO DE APLICACIONES MULTIPLATAFORMA

UD 5. Programación orientada a objetos. Introducción.

Módulo: Programación



Centro de Enseñanza
Gregorio Fernández

Métodos

- Un método es un conjunto de instrucciones que realizan una tarea concreta y que se agrupan bajo un identificador.
- Un método puede ser visto como una caja negra, con unas entradas sobre las que realiza unas operaciones y una salida que genera como resultado.
- Para ejecutar las instrucciones de un método hay que **llamar** al método, escribiendo su identificador en el código fuente del programa.
- La utilidad principal de los métodos es la **reutilización de código**, ya que un método puede ser utilizado muchas veces desde un mismo programa o desde otros programas sin tener que volver a escribir su código.
- Otra utilidad es **mejorar la legibilidad del código**.



Centro de Enseñanza
Gregorio Fernández

Métodos

- Sintaxis de un método:

```
[visibilidad] [static] tipo_retorno nombre ([parámetros])  
{  
    //cuerpo del método  
}
```

- La **visibilidad** es alguno de los modificadores de acceso vistos.
- El método puede ser o no **estático**.
- El **tipo de retorno** se refiere al tipo de datos devuelto por el método, o **void** si no devuelve nada.
- La lista de **parámetros** se indica de la siguiente forma:

tipo1 param1, tipo2 parm2, ...



Centro de Enseñanza
Gregorio Fernández

Métodos

I – Sentencia return

- Sirve para devolver un valor desde el método.
- El valor puede ser un literal, una variable o el resultado de una expresión.
- El valor retornado debe ser del mismo tipo que el especificado en la definición del método.
- Se pueden escribir varias sentencias **return** en el interior del método y en distintas posiciones, aunque hay que tener cuidado de no dejar algún fragmento de código inalcanzable.
- Ejemplo:

```
public int suma(int a, int b) {  
    int resultado = a + b;  
    return resultado;  
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

II – Llamada

- Para hacer uso del método hay que **invocarlo (llamarlo)**.
- Esto se hace escribiendo en el código fuente el nombre del método seguido de (entre paréntesis) tantos parámetros y del mismo tipo como los que se especifiquen en la definición del método.
- Si no hay lista de parámetros se pondrán los paréntesis vacíos.
- En el momento de la llamada al método, el flujo de ejecución del programa pasa de la sentencia de llamada al interior del método, y éste retiene el flujo hasta que se produce el retorno mediante una instrucción return o hasta que termina el cuerpo del método.
- Posteriormente, el flujo de ejecución retorna de nuevo a la línea siguiente desde la cual se llamó al método, continuándose la ejecución del programa.



Centro de Enseñanza
Gregorio Fernández

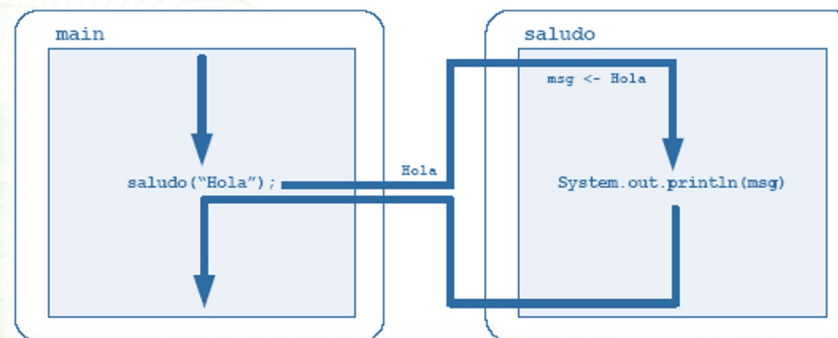
Métodos

II – Llamada

- Veamos gráficamente el proceso de llamada al método **saludo**:

```
public static void main(String[] args) {  
    saludo(Hola);  
}
```

```
public static void saludo(String msg) {  
    System.out.println(msg);  
}
```



gf

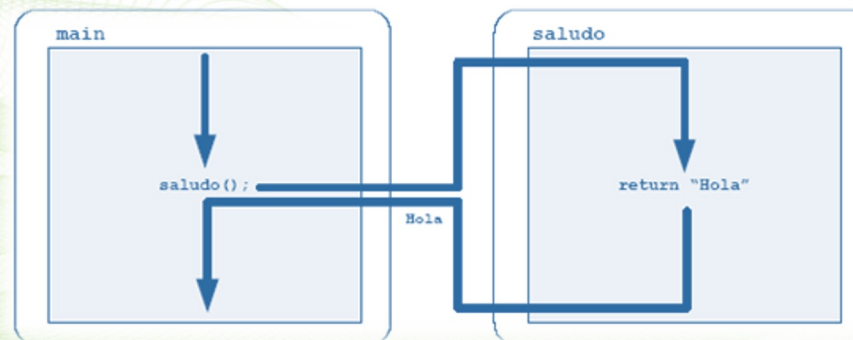

Centro de Enseñanza
Gregorio Fernández

Métodos

II – Llamada

- Modificamos el método **saludo** de la siguiente forma:

```
public static void main(String[] args) {  
    String msg = saludo();  
    System.out.println(msg);  
}  
  
public static String saludo() {  
    return "Hola";  
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

III – Paso de parámetros

- **Parámetros formales**: los definidos en la declaración de un método.
- **Parámetros actuales**: los pasados en la llamada al método.
- Cuando se invoca a un método, los parámetros actuales **se copian** en los parámetros formales.
- Estos dos tipos de parámetros se encuentran en ámbitos diferentes.
- Analicemos el siguiente código:



Centro de Enseñanza
Gregorio Fernández

Métodos

III – Paso de parámetros

```
public double raizCuadrada(double n){  
    if(n>0){  
        return Math.sqrt(n);  
    }  
    System.out.println("Error. Número negativo");  
    return -1;  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    double num=10;  
    double raiz=app.raizCuadrada(num);  
    n=123;  
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

III – Paso de parámetros

- Hacemos un cambio:

```
public double raizCuadrada(double n){  
    n=--num;  
    if(n>0){  
        return Math.sqrt(n);  
    }  
    System.out.println("Error. Número negativo");  
    return -1;  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    double num=10;  
    double raiz=app.raizCuadrada(num);  
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

III – Paso de parámetros

- Y con este cambio, cuanto vale la variable **raiz**?

```
double num=10;
public double raizCuadrada(double n){
    n=--num;
    if(n>0){
        return Math.sqrt(n);
    }
    System.out.println("Error. Número negativo");
    return -1;
}

public static void main(String[] args) {
    Pruebas app=new Pruebas();
    double raiz=app.raizCuadrada(num);
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

III – Paso de parámetros

- Java admite dos formas de pasar parámetros en las llamadas a los métodos dependiendo del **tipo de los parámetros**:
- Paso de parámetros **por valor** → **tipos primitivos**
 - Se pasa una **COPIA DEL VALOR** de los parámetros actuales a los parámetros formales.
 - La información estará duplicada en memoria.
 - Las modificaciones realizadas en el método no afectan a los parámetros actuales.
- Paso de parámetros **por referencia** → **tipos objeto**
 - Se pasa una **COPIA DE LA DIRECCIÓN DE MEMORIA** del objeto (parámetro actual).
 - Parámetro actual y parámetro formal “apuntan” a la misma dirección de memoria en la cual se está almacenado el objeto.
 - Las modificaciones afectan al parámetro actual.



Centro de Enseñanza
Gregorio Fernández

Métodos

III – Paso de parámetros

- Analicemos el siguiente ejemplo de **PASO POR VALOR**:

```
public void metodo(int i) {  
    System.out.println("En el método - Antes i=" + i);  
    i++;  
    System.out.println("En el método - Después i=" + i)  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    int n=0;  
    System.out.println("En el main - Antes n=" + n);  
    app.metodo(n);  
    System.out.p)rintln("En el main - Después n=" + n);  
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

III – Paso de parámetros

- Veamos ahora un ejemplo de **PASO POR REFERENCIA**:

```
public void metodo(MiClase obj) {  
    System.out.println("En el método - Antes a=" + obj.a);  
    obj.a++;  
    System.out.println("En el método - Después a=" + obj.a)  
}
```

```
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    MiClase mC=new MiClase();  
    mC.a=0;  
    System.out.println("En el main - Antes a=" + mC.a);  
    app.metodo(mC);  
    System.out.println("En el main - Después a=" + mC.a);  
}
```

```
public class MiClase {  
    public int a;  
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

IV – Recursividad

- Técnica de programación consistente en que un método **se llame a sí mismo**.
- Permite la realización de algoritmos elegantes.
- Veamos un primer ejemplo:

```
public void contar(int i){  
    System.out.print(i + " ");  
    contar(++i);  
}
```

← Llamada recursiva

- Este método es **recursivo** porque se llama a sí mismo.
- Recibe un número entero, lo imprime por pantalla y se vuelve a llamar a sí mismo, pero pasando el valor recibido como parámetro incrementado en 1.
- Para ejecutarlo hay que llamarlo una **primera vez** pasando un valor inicial:

```
contar(0);
```

- Que ocurre al ejecutar?



Centro de Enseñanza
Gregorio Fernández

Métodos

IV – Recursividad

- Para que un método recursivo no deje un programa “colgado” por **recursividad infinita**, es necesario que exista una **condición de finalización**.
- Esta condición (**if**), hará que la secuencia de llamadas recursivas llegue a su fin y que cada método recursivo finalice de manera natural.
- En el ejemplo anterior la condición de finalización podría ser que cuando imprima el número 99 finalice:

```
public void contar(int i){  
    System.out.print(i + " ");  
    if(i==99) return;  
    contar(++i);  
}
```

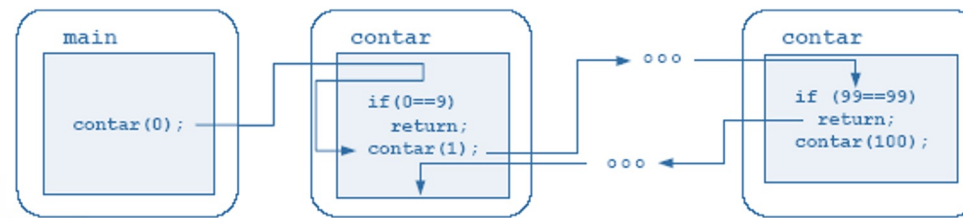


Centro de Enseñanza
Gregorio Fernández

Métodos

IV – Recursividad

- Veamos la secuencia de llamadas recursivas, de forma gráfica:



- En cada una de las llamadas recursivas los métodos quedan “**latentes**” a la espera de que finalice el método al que han llamado, que aunque sea el mismo, es como si fuera un método totalmente diferente.
- Cuando se termina esa espera porque el método invocado finaliza, cada uno de los métodos invocadores continúa su ejecución, de forma que como ya no existen más sentencias terminan de manera natural, retornando el flujo de ejecución al método superior que lo invocó.
- Este proceso de **vuelta a atrás** continúa hasta que el flujo llega al método main y el programa finaliza.

gf

Centro de Enseñanza
Gregorio Fernández

Métodos

IV – Recursividad

- Un ejemplo clásico de recursividad en programación es el algoritmo del cálculo del **factorial** de un número.
- El factorial de un número N (representado por N!), es el producto de todos los números naturales menores o iguales a N:

$$N! = N * (N-1) * (N-2) * \dots * 3 * 2 * 1$$

- Por ejemplo $4! = 4 * 3 * 2 * 1 = 24$.
- El factorial de 0 se define como 1 ($0! = 1$).
- La implementación en java podría ser:

```
public int factorial(int n){  
    if(n==0) return 1;  
    return n * factorial(n-1);  
}
```



Centro de Enseñanza
Gregorio Fernández

Métodos

IV – Recursividad



- Cuestiones a tener en cuenta:

- ✓ Cada llamada recursiva implica que se reserva memoria para las variables del “nuevo” método. Esto implica que si se cae en una recursividad infinita, el programa provocará un error de ejecución por **desbordamiento de la pila** de memoria.
- ✓ Las llamadas recursivas deben hacerse de forma que en cada llamada se resuelva un **problema más sencillo**.
Es más sencillo resolver factorial(4) que factorial(5).
- ✓ Todo problema recursivo debe tener una **condición de finalización** y ésta debe ser **alcanzable** en un número finito de pasos.
- ✓ Todo problema recursivo puede resolverse **de forma no recursiva** (iterativa).
- ✓ En ocasiones usando recursividad, es **más sencillo implementar** soluciones.
- ✓ No olvidar, que es una técnica que **consume memoria**.

A stylized logo consisting of the letters 'gf' in a green, italicized font.

Centro de Enseñanza
Gregorio Fernández

POO - Introducción

- Dos de los paradigmas de programación más difundidos son:
 - Modelo **procedimental** (programación imperativa u orientado a procesos)
 - Modelo **orientado a objetos**.
- El **modelo procedimental** está organizado en torno a los procesos.
- El **modelo orientado a objetos** está orientado a los datos (objetos), los cuales son considerados más importantes que los procesos.
- La potencia de la POO (programación orientada a objetos) está en que hace más sencillo resolver problemas, al dividir el problema en objetos, de forma que cada objeto funciona de forma totalmente independiente.



Centro de Enseñanza
Gregorio Fernández

Objetos

- Java es un lenguaje totalmente orientado a objetos, de forma que el objeto es la entidad fundamental del lenguaje.
- Los objetos son creados a partir de una **clase**.
- Una clase, se puede ver como el tipo de datos del objeto.
- Características de un objeto:
 - **Identidad**. Cada objeto es único y diferente.
 - **Estado**. El estado de un objeto son los valores de sus atributos en un momento determinado.
 - **Comportamiento**. Son los métodos del objeto.



Centro de Enseñanza
Gregorio Fernández

Objetos

- El ciclo de vida de un objeto consta de las siguientes fases:

1. Creación.
2. Uso.
3. Destrucción.



Centro de Enseñanza
Gregorio Fernández

Objetos

I - Creación

- Declarar + Instanciar.
- La **declaración** de un objeto consiste en la declaración de una variable del tipo de la clase del objeto → **referencia**
- Después de declarar el objeto, hay que instanciarlo para poder usarlo.
- La **instanciación** consiste en la creación en sí del objeto, es decir, ponerlo en memoria.
- Para instanciar un objeto se necesita llamar a un **constructor** de la clase mediante el operador **new**.
- Si intentamos usar un objeto sin antes haberlo instanciado se lanza la excepción **NullPointerException**.



Centro de Enseñanza
Gregorio Fernández

Objetos

II - Uso

- Una vez creado el objeto podremos usarlo:
 - Accediendo a sus variables (de instancia).
 - Invocando a sus métodos.
- Para referirse a los miembros de un objeto se utiliza el operador punto “.”:

referencia.miembro



Centro de Enseñanza
Gregorio Fernández

Objetos

III - Destrucción

- Los destructores de objetos se utilizan generalmente, para liberar recursos y cerrar flujos abiertos.
- No reciben parámetros.
- No está permitida su sobrecarga.
- En Java no hay destructores como en otros lenguajes de programación.



Centro de Enseñanza
Gregorio Fernández

Beneficios de la POO

- **Modularidad.** El código fuente de un objeto puede mantenerse y reescribirse sin que haya que reprogramar el código de otros objetos de la aplicación.
- **Reutilización de código.** Podemos utilizar clases y objetos de terceras personas. No tenemos que conocer los detalles de su implementación interna sino solamente su interfaz.
- **Facilidad de testeo y reprogramación.** Si tenemos un objeto que está dando problemas en una aplicación, no tenemos que reescribir el código de toda la aplicación, sino que tenemos que reemplazar el objeto problemático por otro similar, o bien reprogramarlo.
- **Ocultación de la información.** Se ocultan los detalles de implementación. Lo que interesa es la interfaz.



Centro de Enseñanza
Gregorio Fernández

Propiedades de la POO

- **Abstracción.** Según la RAE, abstraer es:

“separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción”.

Cuando se programa orientado a objetos lo que se hace es abstraer las características de los objetos que van a formar parte del programa, y crear las clases con sus atributos y sus métodos.

- **Encapsulamiento.** Los objetos se ven según su comportamiento externo. Podemos ejecutar los métodos de un objeto sin tener por qué saber cómo funciona internamente el método.



Propiedades de la POO

- **Ocultación de la información.** Un objeto está aislado del exterior, de forma que su zona privada sólo es utilizada por métodos de la propia clase, y expone su zona pública (llamada interfaz de la clase) para que pueda ser utilizada por otros objetos.
- **Herencia.** Las clases se estructuran formando jerarquías, de tal forma, que una clase (subclase) puede heredar propiedades de otra clase (superclase).
- **Polimorfismo.** El polimorfismo permite crear métodos compatibles con objetos de distintos tipos. Es un concepto íntimamente relacionado con la herencia.



Clases

- Una clase es la **definición** de un tipo de objeto.
- **Prototipo o plantilla** que define las propiedades y métodos comunes a todos los objetos de esa clase, a partir de la cual se crean objetos.
- A partir de una clase podremos crear objetos de esa clase.
- Una clase es un concepto **abstracto** que no ocupa memoria. Un objeto, sin embargo, es un concepto **real**. Cada objeto tiene su propio espacio en memoria para almacenar sus datos y sus métodos.



Centro de Enseñanza
Gregorio Fernández

Clases

I - Creación

- Definir:
 - **Atributos (campos ó datos).** Es decir, los datos miembros de esa clase. Pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase).
 - **Métodos.** Acciones (operaciones) que puede realizar la clase.
 - **Código de inicialización.** Operaciones previas a la construcción de la clase (método constructor).



Centro de Enseñanza
Gregorio Fernández

Clases

I - Creación

- Sintaxis:

```
[acceso] class nombreClase {
```

```
    [acceso] [static] tipo atributo1;
```

```
    [acceso] [static] tipo atributo2;
```

```
    [acceso] [static] tipo atributo3;
```

```
    ...
```

```
    [acceso] [static] tipo método1([listaDeArgumentos]) {
```

```
        .....
    }
```

```
    ...
}
```

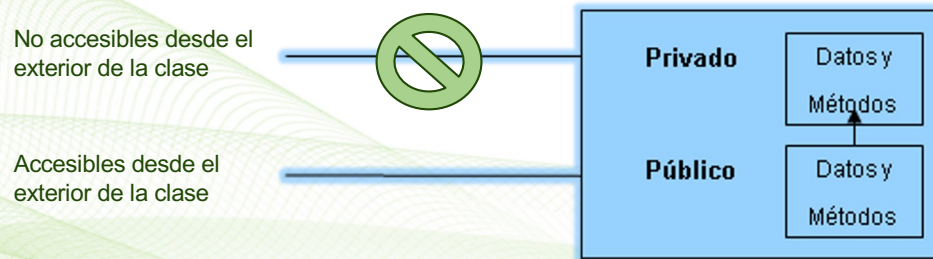


Centro de Enseñanza
Gregorio Fernández

Clases

II - Encapsulamiento

- Uno de los principios fundamentales de la POO es la **ocultación de la información**, lo cual significa que no es posible acceder a miembros de la clase desde el exterior.
- Esto se consigue haciendo esos miembros privados.



Clases

II - Encapsulamiento

- Para controlar el acceso a los miembros de una clase, se utilizan los modificadores de acceso **public**, **private** y **protected**. Éstos, determinan la visibilidad de los miembros de la clase:

Acceso	Miembro de la misma clase	Miembro de una clase derivada	Miembro de la clase del paquete	Miembro de la clase de otro paquete
private	✓	x	x	x
ninguno	✓	x	✓	x
protected	✓	✓	✓	x
public	✓	✓	✓	✓

- A los miembros **public** se puede acceder por cualquier método desde fuera de la clase.
- A los miembros **private** sólo se puede acceder por métodos de la misma clase.
- Y a los miembros **protected**, se puede acceder por métodos de la misma clase ó de clases derivadas, así como por métodos de otras clases que se encuentran en el mismo paquete (se verá con más detalle cuando hablemos de la “herencia”).

gf

Centro de Enseñanza
Gregorio Fernández

Clases

II - Encapsulamiento

- Principio de **ocultación de la información**:

*Toda interacción con un objeto se debe restringir a utilizar una **interfaz** bien definida, que permita que los detalles de implementación de los objetos sean ignorados.*

*Por consiguiente, los **elementos públicos** forman la interfaz externa del objeto, mientras que los **elementos privados** son aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.*



Centro de Enseñanza
Gregorio Fernández

Referencia “this”

- Referencia al objeto con el que se está trabajando.
- Es decir, se refiere al objeto que está ejecutando el método.
- En algunas ocasiones sirve para resolver ambigüedades o para devolver referencias al propio objeto.
- Lo vemos en el siguiente ejemplo:



Centro de Enseñanza
Gregorio Fernández

Referencia “this”

```
public class Rectangulo {  
  
    private int ancho;  
    private int alto;  
  
    public Rectangulo(int ancho, int alto){  
        this.ancho=ancho;  
        this.alto=alto;  
    }  
  
    public int getAncho(){  
        return this.ancho;  
    }  
  
    public int getAlto(){  
        return this.alto;  
    }  
  
    public Rectangulo incrementarAncho(){  
        this.ancho++;  
        return this;  
    }  
  
    public Rectangulo incrementarAlto(){  
        this.alto++;  
        return this;  
    }  
}
```


La clase Object

- Raíz jerárquica de Java.
- Cualquier clase es subclase de **Object**.
- Dispone de una serie de métodos que pueden ser utilizados por todos los objetos, ya que los heredan.
- Consultar el API de java.
- Nosotros vamos a con más profundidad los métodos siguientes:
 - `toString()`
 - `equals()`
 - `getClass()`




Centro de Enseñanza
Gregorio Fernández

La clase Object

I – Método toString()

- Devuelve la representación **textual** del objeto.
- Normalmente las clases redefinen este método para obtener el resultado esperado.
- Ejemplo:

```
Rectangulo r1=new Rectangulo(5,7);  
System.out.println(r1.toString());
```



¿Es lo que
esperaba?



gf

Centro de Enseñanza
Gregorio Fernández

La clase Object

II – Comparar objetos

- Podemos usar el método `equals()`.
- Ejemplo:

```
Rectangulo r1=new Rectangulo(5,3);  
Rectangulo r2=new Rectangulo(5,3);  
System.out.println(r1.equals(r2));
```



- Y ahora?:

```
r1=r2;  
System.out.println(r1.equals(r2));
```



Centro de Enseñanza
Gregorio Fernández

La clase Object

II – Comparar objetos

- Si queremos hacer una comparación de objetos “en profundidad”, tenemos que sobrescribir el método **equals** y determinar el criterio de comparación.
- En nuestro ejemplo:

```
public boolean equals(Rectangulo r) {  
    return r.getAlto()==this.alto &&  
           r.getAncho()==this.ancho;  
}
```



Centro de Enseñanza
Gregorio Fernández

La clase Object

II – Comparar objetos

- Si con el método equals no tenemos suficiente, podemos comparar objetos con el método `compareTo()`.
- Si por ejemplo quisiéramos ordenar “Rectángulos”.
- Para ello, la clase debe implementar la interfaz **Comparable**:

```
public class Rectangulo implements Comparable<Rectangulo>
```

- Y posteriormente, implementar el método compareTo:

```
    public int compareTo(Rectangulo o) {  
        if (this.equals(o)) {  
            return 0;  
        }  
        if (this.ancho > o.getAncho() && this.alto > o.getAlto()) {  
            return 1;  
        }  
        return -1;  
    }
```



Centro de Enseñanza
Gregorio Fernández

La clase Object

III – Método getClass()

- Devuelve la **clase del objeto** en tiempo de ejecución .
- Concretamente, devuelve un objeto **Class** al que se le puede pedir información sobre la clase: nombre, quién es su superclase,...

```
public void mostrarInf(Object obj) {  
    System.out.println("Nombre de la clase: " + obj.getClass().getName());  
    System.out.println("Superclase: " + obj.getClass().getSuperclass());  
}
```

- Un uso práctico del método getClass(), podría ser crear un ejemplar de una clase sin conocer la clase en tiempo de compilación.

```
public Object crearInstanciaDe(Object obj) {  
    return obj.getClass().newInstance();  
}
```



Centro de Enseñanza
Gregorio Fernández

Constructores



- Un **constructor** es un método no estático que se ejecuta automáticamente cuando se crea un objeto de la clase. Sirve para **inicializar** los datos de la clase.
- Reglas:
 - Método no estático.
 - Tiene el mismo nombre que la clase.
 - Puede tener cero o más argumentos.
 - No devuelve ningún valor (ni siquiera void).
 - Debe ser público.



Centro de Enseñanza
Gregorio Fernández

Constructores

I – Por defecto

- Constructor que **no tiene parámetros**.
- Normalmente da valor inicial a los datos de la clase asignándoles valores por defecto.
- Ejemplo:

```
public class Punto {  
    private double x;  
    private double y;  
  
    public Punto() {  
        x=0.0;  
        y=0.0;  
    }  
}
```

- Java crea automáticamente un constructor por defecto cuando no existen otros constructores. Inicializa los datos numéricos (int, float,...) a cero, las variables de tipo boolean a false y las referencias a null.

Constructores

II – Alternativos

- Un constructor con parámetros se denomina **constructor alternativo**.
- Por ejemplo, un constructor alternativo para la clase Punto sería:

```
public class Punto {  
    private double x;  
    private double y;  
  
    public Punto(double _x, double _y) {  
        x=_x;  
        y=_y;  
    }  
}
```

- Ahora para crear objetos Punto tenemos que dar valores a sus coordenadas x e y:

```
Punto p = new Punto(5.0, 47.7);
```

- **OJO:** Si se define una clase con un solo constructor con argumentos y se omite el constructor sin parámetros, ya no será posible utilizar el constructor por defecto.



Constructores

III – Sobrecargados

- Es posible **sobrecargar un constructor**, es decir, definir en su clase más de un constructor, cada uno de ellos con una **firma de argumentos distinta** (distinto número de argumentos o argumentos de distintos tipos).
- Los constructores sobrecargados son bastante frecuentes, ya que proporcionan formas alternativas de crear objetos de una clase.
- Sólo un constructor se ejecuta cuando se crea un objeto, con independencia de cuántos constructores se hayan definido.



Centro de Enseñanza
Gregorio Fernández