

Métodos static, sobreescritura y sobrecarga, constructores.

[Google docs](#)
[pdf](#)

Puntos a tratar.

- Desarrolle código que declare métodos **static** y **no static**, y si es apropiado use nombres de métodos que se apeguen a los estándares de JavaBeans. Desarrolle código que declare y use una lista de argumentos de tamaño variable.
- Dando un ejemplo, determine si un método está **sobreescribiendo** o **sobrecargando** correctamente a otro método, identifique valores de **retorno** válidos para un método.
- Dando un conjunto de clases y superclases, desarrolle **constructores** para una o más clases. Dando una declaración de una clase, determinar si un constructor por defecto debe ser creado, si es así, determine el comportamiento de este constructor. Dando una clase anidada o no anidada, escriba código para instanciar dicha clase.

Introducción.

Repasando: un programa básicamente se conforma por 3 elementos:

- Datos
- Operaciones sobre los datos
- La lógica que determina las operaciones

En Java las clases juntan en una misma entidad los datos y las operaciones sobre ellos. Una clase se forma de variables y métodos.

Una parte importante de la POO es la herencia. Podemos heredar de una clase y la clase derivada hereda los atributos y métodos no privados de la clase padre. Para mantener esto sencillo, Java permite sólo herencia simple, sin embargo para solucionar las implicaciones de esto existen las interfaces.

Sintaxis básica de un método:

```
<modifier> <returnType> <methodName> ( <Parameters>) {  
// body of the method. The code statements go here.  
}
```

Como ejemplo podemos tener:

```
public int square (int number) {  
    return number*number;  
}
```

Usando el método:

```
int myNumber = square(2); //Donde myNumber almacenaría un 4
```

Métodos y variables static.

Los metodos y variables static son compartidos por todas las instancias de una clase y se declaran usando el modificador static.

El modificador static puede ser aplicado a una variable, un método y un bloque de código.

Debido a que un elemento static es visible por todas las instancias, cuando se efectúe un cambio todas las instancias verán el cambio.

Ejemplo 1.

```
class StaticExample {
    static int staticCounter=0;
        int counter=0;
    StaticExample() {
        staticCounter++;
        counter++;
    }
}
```

```
class RunStaticExample {
    public static void main(String[] args) {
        StaticExample se1 = new StaticExample();
        StaticExample se2 = new StaticExample();
        System.out.println("Value of staticCounter for se1: " +
se1.staticCounter);
        System.out.println("Value of staticCounter for se2: " +
se2.staticCounter);
        System.out.println("Value of counter for se1: " +
se1.counter);
        System.out.println("Value of counter for se2: " +
se2.counter);

        StaticExample.staticCounter = 100;
        System.out.println("Value of staticCounter for se1: " +
se1.staticCounter);
        System.out.println("Value of staticCounter for se2: " +
se2.staticCounter);
    }
}
```

Este código produce la salida:

```
Value of staticCounter for se1: 2  
Value of staticCounter for se2: 2  
Value of counter for se1: 1  
Value of counter for se2: 1  
Value of staticCounter for se1: 100  
Value of staticCounter for se2: 100
```

Precaución: Una variable static se inicializa cuando se carga una clase, mientras que una variable de instancia cuando un objeto es creado.

Al igual que los atributos un método static pertenece a la clase no a una instancia. Un método static no puede acceder a variables o métodos no-static. Debido a que los métodos static no pertenecen pertenecen a toda la clase pueden ser ejecutados incluso sin haber creado alguna instancia. Por ejemplo, todas las clases contienen el método static llamado main que se ejecuta sin necesidad de instanciar la clase.

Ejemplo:

```
class MyClass {
    String salute = "Hello";
    public static void main(String[] args){
        System.out.println("Salute: " + salute);
    }
}
```

No compila por que un método static trata de acceder a un atributo no-static.

Se dice que una clase consiste en variables y métodos, esto es cierto la mayoría de las veces, sin embargo una clase puede contener un bloque de código static afuera de un método, es decir que no pertenece a ningún método.

Por ejemplo podría necesitar ejecutar cierta tarea antes de que la clase sea instanciada, o antes de usar el método main.

```
class StaticCodeExample {
    static int counter=0;
    static {
        counter++;
        System.out.println("Static Code block: counter: " + counter);
    }
    StaticCodeExample() {
        System.out.println("Construtor: counter: " + counter);
    }
    public static void main(String[] args) {
        System.out.println("Main method");
    }
}
```

```
public class RunStaticCodeExample {
    public static void main(String[] args) {
        System.out.println("Start");
        StaticCodeExample sce = new StaticCodeExample();
        System.out.println("main: counter: " +
sce.counter);
    }
}
```

Si ejecutamos la primer clase obtenemos:

```
Static Code block: counter: 1  
Main method
```

Debido a que primero se carga la variable static después se ejecuta el bloque de código static y finalmente el main.

Ahora bien si ejecutamos el segundo ejemplo obtenemos:

```
Start  
Static Code block: counter: 1  
Constructor: counter: 1  
main: counter: 1
```

En el momento en que se carga la clase StaticCodeExample esta carga la variable static y ejecuta el bloque de código static, luego entra el constructor.

El código static es ejecutado sólo una vez al momento de cargar la clase, independientemente de donde se encuentre, el siguiente ejemplo no alteraría el funcionamiento del anterior:

```
class StaticCodeExample {
    static int counter=0;

    StaticCodeExample() {
        System.out.println("Construtor:  counter: " + counter);
    }
    public static void main(String[] args) {
        System.out.println("Main method");
    }

    static {
        counter++;
        System.out.println("Static Code block: counter: " + counter);
    }
}
```

Métodos con número variable de parámetros.

Asuma que necesita crear un método que pueda ser llamado con cualquier número de argumentos, por ejemplo:

```
myMethod(1);  
myMethod(1, 2);  
myMethod(1, 2, 3);
```

En versiones previas de Java necesitaríamos crear 3 métodos distintos (como se muestra en el ejemplo) □

Ejemplo:

```
class MyClass {
    public void printStuff(String greet, int... values) {
        System.out.println("Start");
        for (int v : values ) {
            System.out.println( greet + ":" + v);
        }
    }
}
```

```
public class VarargTest {
    public static void main(String[] args) {
        MyClass mc = new MyClass();
        mc.printStuff("Hello", 1);
        mc.printStuff("Hey", 1,2);
        mc.printStuff("Hey you", 1,2,3);
        mc.printStuff("Hi");
    }
}
```

Produce la siguiente salida:

Start

Hello:1

Start

Hey:1

Hey:2

Start

Hey you:1

Hey you:2

Hey you:3

Start

El programa invoca al método `printStuff(...)` que recibe un número variable de argumentos de tipo entero.

Un error en el compilador lo causaría sustituir la declaración del método por:

```
public void printStuff(int... values, String greet) {
```

Recordemos que debe ser el último el parámetro de tamaño variable.

Además, sólo puede haber uno variable, la siguiente línea también sería errónea:

```
public void printStuff(String greet, int... values, double... dnum) {
```

Otro ejemplo lo tenemos a continuación:

```
public class VarargsTest
{
    // calculate average
    public static double average( double ... numbers )
    {
        double total = 0.0; // initialize total

        // calculate total using the enhanced for statement
        for ( double d : numbers )
            total += d;

        return total / numbers.length;
    } // end method average

    public static void main( String args[] )
    {
        double d1 = 10.0;
        double d2 = 20.0;
        double d3 = 30.0;
        double d4 = 40.0;
```

```
        System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 =  
%.1f\n\n", d1, d2, d3, d4 );
```

```
        System.out.printf( "Average of d1 and d2 is %.1f\n", average(  
d1, d2 ) );
```

```
        System.out.printf( "Average of d1, d2 and d3 is %.1f\n",  
average( d1, d2, d3 ) );
```

```
        System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",  
average( d1, d2, d3, d4 ) );
```

```
    } // end main
```

```
} //
```

Nombrando métodos de acuerdo a las reglas de JavaBeans

Un JavaBean es un tipo de clase Java que es definida respetando ciertas reglas, incluyendo las convenciones de nombramiento para sus variables y métodos. Las reglas son las siguientes:

- Las variables private de un JavaBean, llamadas propiedades, sólo pueden ser accedidas por medio de métodos getter y setter. Las propiedades se nombran por ejemplo: myProperty.
- Todas las propiedades no-boolean tienen un método getter usado para obtener el valor de la propiedad. se nombran por ejemplo: getMyProperty().
- Todas las propiedades tienen un método setter usado para modificar el valor de una propiedad. Ejemplo setMyProp().
- Los métodos setter y getter deben ser public, para que cualquiera que use el bean pueda invocarlos.
- Un método setter regresa void y tiene un parámetro que representa el tipo de la propiedad.
- Un método getter no recibe parámetros y regresa el tipo de dato correspondiente al que recibe el setter.

Ejemplo:

```
public class ScoreBean {
    private double meanScore;
    // getter method for property meanScore
    public double getMeanScore() {
        return meanScore;
    }
    // setter method to set the value of the property meanScore
    public void setMeanScore(double score) {
        meanScore = score;
    }
}
```

Note los nombres y la variable a la que se refieren.

Clases anidadas.

Todas las clases vistas hasta ahora entran en la categoría de top-level. Java permite definir clases dentro de una clase top-level, se les llama anidadas.

Estas clases se verían así:

```
class <OuterClassName> {  
    // variables and methods for the outer class  
    ...  
    class <NestedClassName> {  
        // variables and methods for the nested class  
        ...  
    }  
}
```

Como sabemos los elementos de una clase pueden ser static y no-static.

Una clase no-static anidada se le llama clase interna.

Una clase static anidada simplemente se le dice anidada.

Características:

- Una clase interna (clase no-static) es asociada con una instancia de la clase contenedora.
- Una clase interna tiene acceso a todas las partes de la clase contenedora, incluso los private.
- Al igual que variables y métodos static una clase static refiere a toda la clase, no a una instancia, de igual manera la clase anidada static no puede acceder a los elementos no-static de la clase contenedora.
- Una clase interna por ser no-static es asociada sólo con una instancia de la clase contenedora y por lo tanto no se pueden definir partes static dentro de una clase interna.
- Una instancia de una clase interna sólo puede existir dentro de una instancia de la clase contenedora, justo como otro miembro no-static de la clase contenedora.
- Las clases anidadas pueden ser declaradas abstract o final, con el mismo significado.
- Los modificadores de acceso como public, private o protected pueden ser usados por clases internas y tendrán el mismo significado que otros elementos.
- Las clases anidadas pueden ser declaradas en cualquier bloque de código y tendrán acceso a todas las variables dentro del bloque de código.

Ejemplo:

```
class TestNested {
    public static void main(String[] args) {
        String ext = "From external class";
        MyTopLevel mt = new MyTopLevel();
        mt.createNested();
        MyTopLevel.MyInner inner = mt.new MyInner();
        inner.accessInner(ext);
    }
}

class MyTopLevel{
    private String top = "From Top level class";
    MyInner minn = new MyInner();
    public void createNested() {

        minn.accessInner(top);
    }
}
```

```
class MyInner {  
    public void accessInner(String st) {  
        System.out.println(st);  
    }  
}  
}
```

Producirá:

```
From Top level class
```

```
From external class
```

Note como es instanciada la clase interna en la clase externa:

```
MyTopLevel mt = new MyTopLevel();
```

```
MyTopLevel.MyInner inner = mt.new MyInner();
```

Algo similar podría lograrse con:

```
MyTopLevel.MyInner inner = new MyTopLevel().new MyInner();
```

Ahora bien, una clase anidada static:

```
class TestStaticNested {
    public static void main(String[] args) {
        String ext = "From external class";
        new MyTopLevel().gateToStatic();
        MyTopLevel.StaticNested sn = new MyTopLevel.StaticNested();
        sn.accessStaticNested(ext);
    }
}

class MyTopLevel{
    private static String top = "From top level class";

    public static void gateToStatic(){
        StaticNested s = new StaticNested();
        s.accessStaticNested(top);
    }

    static class StaticNested {
```

```
public void accessStaticNested(String st) {  
    System.out.println(st);  
}  
}  
}
```

Constructores.

Cuando instanciamos una clase el objeto se almacena en memoria. Dos elementos están asociados a este proceso el operador new y los métodos especiales llamados constructores.

El constructor tiene el nombre de la clase y no especifica valor de retorno. Cuando el JRE encuentra una sentencia new lo mapea en memoria y después ejecuta el constructor para inicializar ese espacio en memoria.

Ejemplo:

```
ComputerLab csLab = new ComputerLab( ) ;
```

- 1.- Localiza espacio en memoria para la instancia csLab.
- 2.- Inicializa las variables de csLab.
- 3.- Ejecuta el constructor ComputerLab().

Si no se especifica un constructor el compilador provee uno llamado constructor default, si por el contrario existe al menos uno definido el compilador no ya no crea alguno.

- Afuera de la clase, el constructor sólo puede ser llamado por medio de la sentencia new.

- Adentro de la clase un constructor puede llamarse desde otro constructor únicamente.

- Con this podemos llamar a otro constructor de la misma clase.

- Con super llamaremos constructores de la clase padre llamada también super-clase.

En ambos casos siempre es la primera línea del constructor.

- Si no se incluye una llamada con super o con this el compilador pone una llamada super al principio del constructor

- Si se agrega una llamada super el compilador ya no interfiere.

Ejemplo:

```
class TestConstructors {
    public static void main(String[] args) {
        new MySubSubClass();
    }
}

class MySuperClass {
    int superVar = 10;
    MySuperClass(){
        System.out.println("superVar: " + superVar);
    }
    MySuperClass(String message) {
        System.out.println(message + ": " + superVar);
    }
}
```

```
class MySubClass extends MySuperClass {
    int subVar = 20;
    MySubClass() {
        super("non default super called");
        System.out.println("subVar: " + subVar);
    }
}
```

```
class MySubSubClass extends MySubClass {
    int subSubVar = 30;
    MySubSubClass() {
        this("A non-deafult constructor of MySubSubClass");
        System.out.println("subSubVar: " + subSubVar);
    }
}
```

```
MySubSubClass(String message){
    System.out.println(message);
}
}
```

Note que MySubSubClass hereda de MySubClass y esta a su vez de MySuperClass, si compilamos obtenemos:

```
non default super called:10
```

```
subVar: 20
```

```
A non default constructor of MySubSubClass
```

```
subSubVar: 30
```

Al ejecutar la tercera línea el constructor MySubSubClass() es llamado y el llama a MySubSubClass(String message).

Como este constructor no especifica ninguna llamada el compilador coloca una llamada super().

A continuación se ejecuta el constructor MySubClass() que especifica la llamada al constructor MySuperClass(String message)

Ahora bien, el compilador puede llevarnos a un error de la siguiente manera:

```
class A {
    int myNumber;
    A(int i) {
        myNumber = i;
    }
}
class B extends A {
    String myName;
    B (String name) {
        myName = name;
    }
}
```

Debido a que el compilador coloca una llamada `super()` al principio del constructor de la clase B, sin embargo el constructor para la clase A recibe un parámetro. Para solucionarlo podríamos agregar un constructor default a la clase A.

Sobreescritura y sobrecarga de métodos.

Son dos características notables de Java.

- Sobreescribir permite modificar el funcionamiento de un método heredado.
- Sobrecargar permite usar el mismo método con diferentes objetivos.

Estas dos características dan facilidad, extensibilidad y flexibilidad al código de Java.

Sobreescritura.

- No puede sobreescribir un método que es declarado final.
- No puede cambiar un método declarado static para hacerlo no-static.
- El sobreescrito debe tener el mismo valor de retorno.
- Los parámetros deben ser el mismo número y del respectivo tipo, en el mismo orden.
- No puedes sobreescribir un método para hacerlo menos accesible. Ejemplo generará error sobreescribir un método public y declararlo protected. Lo contrario es posible.
- Si el método a sobreescribir tiene una declaración throws:
 - * El método sobreescrito tendrá throws también.
 - * Cada excepción incluida en la sentencia throws deberá estar en el método sobreescrito o una subclase de este.
- Si el método sobreescrito tiene throws el método a sobreescribir no tiene que tenerlo.

Ejemplo:

```
protected int aMethod(String st, int i, double number);
```

Sobreescritura del método válidas:

<pre>protected int aMethod(String st, int i, double number)</pre>	Valid	Same signature
<pre>protected int aMethod(String st, int j, double num)</pre>	Valid	Same signature
<pre>protected double aMethod(String st, int i, double number)</pre>	Invalid	Different return type
<pre>protected int aMethod(int i, different order String st, double number)</pre>	Invalid	Argument types are in
<pre>protected int aMethod(String st, types int i, double number, int j)</pre>	Invalid	Different number of
<pre>protected int aMethod(String st, int i)</pre>	Invalid	Different number of
<pre>int aMethod(String st, int i, less public</pre>	Invalid	Default modifier is

double number)

than protected

Otro ejemplo:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Animal.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}
```

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Cat.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat.");
    }
}
```

```
public static void main(String[] args) {
    Cat myCat = new Cat();
    Animal myAnimal = myCat;
```

```
Animal.testClassMethod();  
myAnimal.testInstanceMethod();  
}
```

Sobrecarga.

Nos ayudará a definir métodos con el mismo nombre.

- Dos métodos sobrecargados no pueden tener una lista idéntica de parámetros.
- Los valores de retorno pueden ser los mismos o diferentes.

Ejemplo:

```
class TestAreaCalculator {
public static void main(String[] args) {
    AreaCalculator ac = new AreaCalculator();
    System.out.println("Area of a rectangle with length 2.0, and width
3.0: " + ac.calculateArea(2.0f, 3.0f));
    System.out.println("Area of a triangle with sides 2.0, 3.0, and
4.0: " + ac.calculateArea(2.0, 3.0, 4.0));
    System.out.println("Area of a circle with radius 2.0: " +
ac.calculateArea(2.0));
}
}
```

```
class AreaCalculator {
```

```
    double s = (a+b+c)/2.0;  
    return Math.sqrt(s*(s-a)*(s-b)*(s-c));  
  }  
}
```

Produce la salida:

```
Area of a rectangle with length 2.0, and width 3.0: 6.0
```

```
Area of a triangle with sides 2.0, 3.0, and 4.0: 2.9047375096555625
```

```
Area of a circle with radius 2.0: 12.566370614359172
```

Un método heredado en una subclase puede ser sobrescrito y sobrecargado, por ejemplo:

```
class VolumeCalculator extends AreaCalculator {  
    int calculateArea (int i, int j) {  
        }  
    double calculateVolume (double x, int y, double z);  
    }  
}
```

El método calculateArea(...) es una versión sobrecargada del método heredado.

Ejemplos de sobrecargas en esta última clase:

<pre>int calculateArea(float a, float b) CalculateArea. But overridden nor compiler will</pre>	<p>Invalid</p> <p>The method has the same name as inherited from the superclass CalculateArea. But this method is neither correctly overridden nor correctly overloaded. Hence the compiler will generate an error on this.</p>
<pre>double calculateVolume different order. (double x, double y, int z) of the existing</pre>	<p>Valid</p> <p>Same set of argument types but in different order. Therefore it is a valid overloading method.</p>
<pre>int calculateVolume same order as the (double x, int z, double y) same name. cannot be</pre>	<p>Invalid</p> <p>Same set of argument types in the same order as the previously defined method with the same name. Therefore, invalid overloading. It cannot be</p>

does not have it.

```
void CalculateVolume()  
from the existing
```

overloading.

Valid

overridden because the superclass

It will generate a compiler error.

Different set of argument types

method. Therefore valid

Sobrecarga de constructores.

Ejemplo:

```
public class ConstOverload{
    public static void main(String[] args) {
        new A();
    }
}
class A {
    int x=0;
    A(){
        this(5);
        System.out.println("A() ");
    }
    A(int i){
        // this();
        System.out.println(i);
    }
}
```

