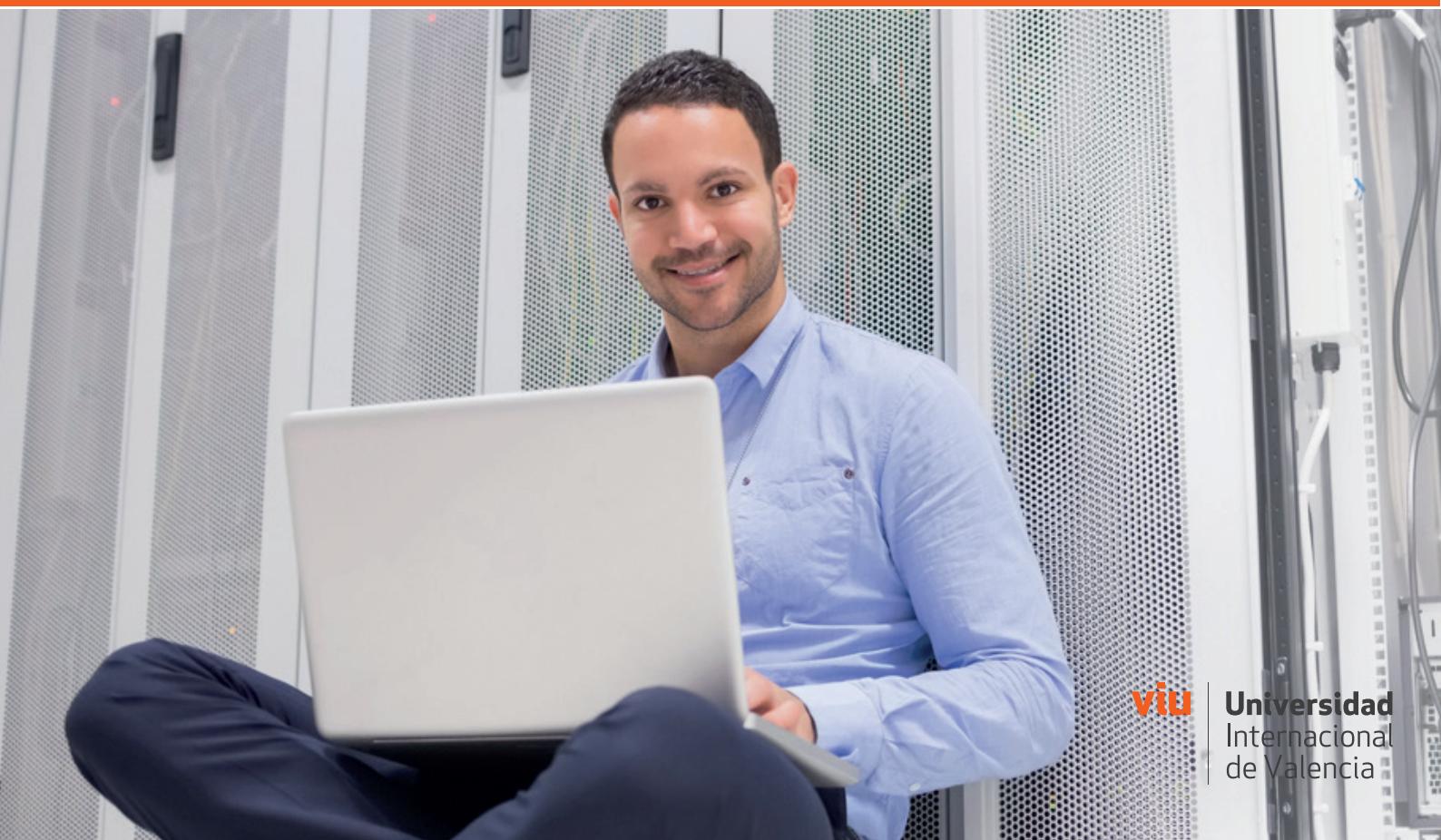


GRADO EN INGENIERÍA INFORMÁTICA

Programación de Computadores

PROYECTOS DE PROGRAMACIÓN

Dr. Roger Clotet Martínez



viu

**Universidad
Internacional
de Valencia**



Este material es de uso exclusivo para los alumnos de la VIU. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la VIU, sin autorización expresa de la misma.

Edita

Universidad Internacional de Valencia

Grado en
Ingeniería Informática

Proyectos de Programación

Programación de Computadores

6 ECTS

Dr. Roger Clotet Martínez

Índice

TEMA1. CONCEPTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS	7
1.1 Abstracción	8
1.2 Clases, objetos y métodos	8
1.3 Herencia.....	12
1.4 Constructores	18
1.5 Interfaces.....	20
1.6 Polimorfismo.....	23
1.7 Elementos estáticos: atributos y funciones	23
1.8 Captura de excepciones.....	24
1.9 Lanzamiento de excepciones	25
1.10 Errores	27
TEMA 2. ESPECIFICACIÓN: DIAGRAMAS DE CLASES Y CASOS DE USO	29
2.1. UML.....	29
2.1.1 Diagramas de clases	30
2.1.2 Casos de uso.....	33
TEMA 3. DISEÑO: ARQUITECTURA EN 3 CAPAS	35
TEMA 4. EL LENGUAJE DE PROGRAMACIÓN JAVA	39
4.1. Máquina Virtual Java	40
4.2. Introducción al lenguaje.....	40
4.3. Punteros y asignación de memoria.....	47
4.4. Sintaxis básica de Java.....	48
TEMA 5. DEPURACIÓN DE PROGRAMAS	51
5.1 Depurador en NetBeans.....	52
TEMA 6. DOCUMENTACIÓN	55
6.1 Javadoc.....	55
TEMA 7. CONCEPTOS BÁSICOS DE DISEÑO DE INTERFACES.....	59
7.1. Java Abstract Window Toolkit (Java AWT)	59

TEMA 8. EJEMPLO: CALCULADORA BÁSICA CON BASE DE DATOS.....	67
8.1. Amazon Web Services Educate.....	67
8.2. Conector MySQL para Java	76
8.3. Código Java	77
GLOSARIO	87
ENLACES DE INTERÉS.....	89
BIBLIOGRAFÍA	91

Leyenda

abc **Glosario**
Términos cuya definición correspondiente está en el apartado “Glosario”.



Enlace de interés
Dirección de página web.

Tema1. Conceptos de programación orientada a objetos

La Programación Orientada a Objetos (POO) es un paradigma de programación basado en el concepto de objetos. Un **objeto** es un contenedor que integra tanto los datos, en forma de atributos; el código necesario para interactuar con los datos (o con una parte de ellos, pues podemos tener algunas restricciones de accesibilidad según como los definamos) y las acciones propias del objeto en forma de funciones/métodos. Algunos objetos pueden contener todos o algunos de los tres elementos, no es necesario tenerlos todos para que sean considerados objetos. Existen diversas aproximaciones a como trabajar con POO, pero la más común y la que veremos en esta asignatura es trabajar con el concepto de **clases** (Llinàs, 2010).

La mayoría de los lenguajes de programación más populares soportan, en mayor o menor grado, el paradigma de la programación orientada a objetos. Por ejemplo: Python, Java, C++, C#, Eiffel, ABAP, Ruby, PHP, Visual Basic .NET, JavaScript, Perl son lenguajes que incorporan elementos de la POO (Fernández, 2012).

En esta asignatura veremos las diferentes características de la POO, y cuando demos ejemplos, mientras no se diga lo contrario, estos estarán basados en Java. Si os perdéis con los ejemplos por no tener las nociones básicas de Java, podéis ir primero al tema *El lenguaje de programación Java* y posteriormente regresar a las nociones de POO para poder seguir más fácilmente los ejemplos.

1.1 Abstracción

Según el diccionario de la Real Academia Española (RAE, <https://dle.rae.es/>), *abstraer: separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción.*

En el caso particular de los lenguajes de programación, hablamos de abstracción cuando capturamos en un objeto abstracto, valga la redundancia, las características esenciales de un determinado conjunto de elementos, las que son comunes a todos ellos y que son relevantes para nuestra implementación. Veámoslo con un ejemplo en la figura 1:

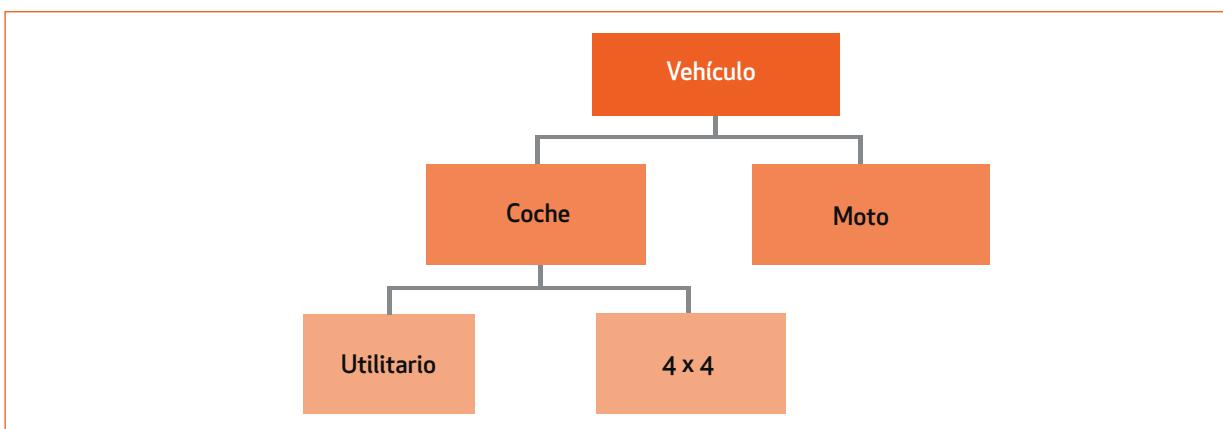


Figura 1. Ejemplo de abstracción. Fuente: elaboración propia.

Tenemos la abstracción vehículo, que en este caso lo podríamos simplificar con los atributos que nos interesan: motor, autonomía, fabricante, modelo, número de plazas, consumo, etc. Después tendríamos un segundo nivel de abstracción, en el cual diferenciaríamos entre coche y motos, por ejemplo, por la cantidad de ruedas. Estos dos objetos, compartirían las características de vehículo, eso es lo que llamamos **herencia** como veremos más adelante. Después podríamos tener un tercer nivel para diferenciar los utilitarios de los 4x4, por ejemplo, por la distancia al suelo, la capacidad de vadeo y la tracción a las cuatro ruedas.

También podríamos hacer otro ejemplo con los animales, y sus subclásificaciones en aves, mamíferos, reptiles... en ese caso la abstracción animales solo tendría las características comunes.

Siempre graduaremos el nivel de detalle y abstracción en función del problema que queremos representar y solucionar. Por ejemplo, si solo queremos saber qué vehículo es más eficiente para transportar personas de A a B, no nos haría falta la marca, con los datos de número de plazas y consumo sería suficiente. Podríamos calcular la eficiencia como:

$$\text{Eficiencia} = \text{número de plazas} / \text{consumo}$$

1.2 Clases, objetos y métodos

Entremos en el tema de la orientación a objetos. Lo primero es definir qué es una **clase**: **una clase es un tipo de datos más las funciones que actúan sobre el mismo**. Adicionalmente cada clase será

una abstracción de los objetos reales que queramos tener en nuestro sistema, cuando a una clase le asignamos valores y se convierte en un objeto "real", decimos que hemos hecho una **instanciación** de la clase.

Veamos de forma práctica con un ejemplo en Java. Definamos una clase que describa la suma de dos números:

```
public class Suma
{
    private double n1;
    private double n2;
    private double res;
    public Suma(double num1, double num2)
    {
        n1=num1;
        n2=num2;
        res=n1+n2;
    }
    public double resultado()
    {
        return res;
    }
    public String toString()
    {
        return n1+" + "+n2+" = "+res;
    }
}
```

Como vemos, se parece bastante a un programa en C o Python. Una de las cosas que primero detectamos son los **privates** y los **publics** delante de las definiciones de las variables y los métodos (o funciones). En este punto entramos en el tema de la visibilidad de los atributos de una clase: las variables y los métodos pueden tener tres tipos de visibilidad:

- **Public:** definimos un atributo como **public** cuando es **visible para todas las clases**. Si es una variable, podemos tanto verla como modificarla desde cualquier clase. No es una buena práctica definir las variables como **public** a menos que haya un buen motivo para ello, o que sea una clase muy simple.
- **Protected:** cuando un atributo es **protected**, **sólo puede ser utilizado dentro del mismo paquete de clases, o desde las subclases de esa clase** (las que la heredan). En estos dos casos se considera como público. Para el resto de clases se trata de un atributo privado.
- **Private:** un atributo **private** sólo es **visible desde dentro de la clase que lo ha declarado**, así que no podremos utilizarlo desde ninguna otra clase.

Analicemos ahora con más detalle el ejemplo de la clase Suma:

```
public class Suma
{
    private double n1;
    private double n2;
    private double res;
    ...
}
```

Esto es la declaración de las variables de la clase. Un objeto de la clase suma tendrá tres atributos: n1, n2, y res, los tres privados y de tipo double. Podríamos inicializarlos a algún valor, pero no es necesario y no lo haremos en este caso.

```
...
public Suma(double num1, double num2)
{
    n1=num1;
    n2=num2;
    res=n1+n2;
}
...
```

Aquí tenemos una función constructora, función especial que se llama igual que la clase y será llamada cuando creemos un objeto de ese tipo (Suma()). Para construir una suma, en este caso, necesitamos dos números, que son los parámetros que le pasamos a la función. La sintaxis es bastante clara: n1, n2 y res se refieren a las variables n1, n2 y res del objeto que acabamos de crear. Si en algún momento quisiéramos referirnos al objeto en sí, podríamos utilizar la palabra clave this. Podemos ver que hemos utilizado nombres de variable diferentes para los parámetros de la función y las variables de la clase para evitar confusiones, pero podríamos haber utilizado los mismos y mediante this podríamos diferenciar si nos referimos a las variables propias de la clase o a los parámetros de la función:

```
...
public Suma(double n1, double n2)
{
    this.n1=n1;
    this.n2=n2;
    res=n1+n2;
}
...
```

Ahora hacemos referencia a la variable n1 del objeto actual (this .n1) y le asignamos la variable n1 que hemos recibido como parámetro de la función. De igual forma para n2.

```
...
public double resultado()
{
    return res;
}
...
```

Esta función apenas merece explicación. Sólo cabe preguntarse si es necesaria, ya que, si nosotros tuviéramos una variable de clase Suma llamada, por ejemplo, `s1`, quizá podríamos hacer referencia a la variable `res` directamente usando `s1.res`. Y hemos dicho quizá, ya que de hecho no podemos. Recordemos que hemos declarado `res` como privada, de manera que no se puede acceder a ella desde fuera de la clase.

Si nos preguntamos por qué no declararla pública, es porque no queremos que se pueda modificar su valor desde fuera, por un lado, y por otro porque el resto de clases no tienen por qué saber cómo está hecha la nuestra por dentro.

Veamos un ejemplo de programa que haga uso de esta clase, para acabar de entender su funcionamiento:

```
public class Sumador
{
    public static void main(String args[])
    {
        // Dos sumas de números.
        Suma s1;
        Suma s2;
        s1=new Suma(3, 4);
        s2=new Suma(5, -6);
        System.out.println(s1.toString());
        System.out.println(s2.toString());
        // Aquí obtenemos los números a partir de dos Strings.
        Double d1, d2;
        double n1, n2;
        d1=new Double("23.5");
        d2=new Double("-4.6");
        n1=d1.doubleValue();
        n2=d2.doubleValue();
        s1=new Suma(n1, n2);
        System.out.println(s1.toString());
    }
}
```

¿Qué hay de interesante en este código? En la primera parte solo tenemos el uso de `new` que nos sirve para crear un objeto nuevo de una determinada clase (ejecutar su función constructora), en la segunda vemos una forma de pasar/convertir cadenas a números. Antes de entrar en detalle, observamos que

se utilizan dos tipos diferentes: `double` y `Double`. Hay que saber que, en la librería `java.lang` hay una clase para cada tipo primitivo de datos de Java, que se llaman igual al tipo, pero con la primera letra en mayúsculas. Se trata de envoltorios/encapsulados para estos tipos de datos, de manera que puedan tener funciones que nos den utilidades que actúen sobre ellos para realizar las operaciones más comunes. Recordemos que una clase es un tipo de datos más un grupo de funciones que actúan sobre estos: los tipos primitivos (`int`, `char`, `double`...) no son clases; sus envoltorios respectivos (`Integer`, `Character`, `Double`...) sí que lo son.

Analicemos esta parte con más detalle:

```
d1=new Double("23.5");
d2=new Double("-4.6");
```

Esto debería entenderse sin demasiados problemas: creamos dos objetos del tipo `Double`. Para hacerlo invocamos la función constructora de esta clase pasándole como parámetro un `String` con el valor en texto plano del real que queremos crear.

```
n1=d1.doubleValue();
n2=d2.doubleValue();
```

Si miramos la documentación para la clase `Double`:



Double

<https://docs.oracle.com/javase/8/docs/api/java/lang/Double.html>

Vemos que tiene una función llamada `doubleValue`, que devuelve un `double` con el valor del objeto. Eso es justamente lo que hacemos en estas dos líneas de código, con lo que hemos conseguido pasar una cadena de caracteres a una variable de tipo `double`.

1.3 Herencia

La orientación a objetos no es un cambio radical respecto a la programación tradicional en muchos aspectos, solo hay que replantearse la forma de estructurar el programa. La potencia, y principal ventaja, de este tipo de lenguajes (los orientados a objetos) viene dada por la herencia y lo que podemos hacer gracias a esta.

La herencia es el medio mediante el cual una clase obtiene todas las características y funcionalidades de otra clase, al mismo tiempo que nos permite extender dichas características y funcionalidades para hacerla más específica.

Veamos exactamente en qué consiste mediante algunos ejemplos: partiendo del ejemplo de la clase `Suma` que ya hemos visto, supongamos que queremos hacer una calculadora. Necesitaremos, además de `Suma`, también las clases `Resta`, `Multiplicacion` y `Division`. Las cuales si lo analizamos brevemente tienen muchos aspectos en común: las cuatro clases son operaciones realizadas sobre dos variables, que regresan un único resultado, y en consecuencias necesitarán atributos y funciones parecidas. Definimos una clase `Operacion` que reúna las partes donde coinciden las cuatro clases y hacemos que cada una de ellas sean subclases de la clase padre `Operacion`. Esta manera de tener

clases y subclases que tienen características comunes es lo que se conoce como herencia, esta relación entre padre o madre e hijos o hijas. El programa podría quedar de la siguiente forma, en primer lugar, la clase Operacion:

```
public abstract class Operacion
{
    protected double n1;
    protected double n2;
    protected double res;
    protected Operacion(double num1, double num2)
    {
        n1=num1;
        n2=num2;
        res=operar();
    }
    protected abstract double operar();
    public abstract char operador();
    public double resultado()
    {
        return res;
    }
    public String toString()
    {
        char op=operador();
        return n1+" "+op+" "+n2+" = "+res;
    }
}
```

Quedando la clase Suma como hija/hijo, en Java utilizamos la palabra clave `extends` para indicar esta relación de herencia:

```
public class Suma extends Operacion
{
    public Suma(double num1, double num2)
    {
        super(num1, num2);
    }
    protected double operar()
    {
        return n1+n2;
    }
    public char operador()
    {
        return '+';
    }
}
```

Si compilamos estas dos clases nuevas, y las copiamos en el directorio donde teníamos la clase Sumador, ¿seguirá funcionando? Pues sí, es una de las particularidades del Java. Podemos modificar una clase internamente, siempre que las clases que la usan no encuentran diferencias “hacia fuera” (nombres de las funciones, parámetros de entrada o salida de cada función), todo seguirá funcionando; que el resultado obtenido sea el mismo dependerá de que sigan realizando el mismo cálculo internamente y no de cómo lo hagan ahora. No tenemos la necesidad de modificar todos los demás elementos del programa.

El Java carga las clases dinámicamente, es decir, cuando tiene que usar una clase, la carga en ese momento a memoria, e invoca los métodos usando su nombre. Eso también significa que, si modificamos una clase, por ejemplo, eliminamos o cambiamos el nombre de alguno de sus métodos que usa otra clase en caliente (mientras ya se está ejecutando el programa principal) podremos seguir ejecutando el programa, hasta que al intentar ejecutar el método modificado se lanzará una excepción de error al no encontrarlo.

En los ejemplo hemos introducido algunos elementos no utilizados hasta ahora, veamos cuales son y porque los usamos:

- **Abstract**

En la definición de la clase utilizamos `abstract`. Esto bloquea la creación de objetos de este tipo. En nuestro ejemplo nunca crearemos operaciones, sino que crearemos sumas, restas... y esta es la forma de decirle a Java que no debe, no se puede, crear instancias de esta clase. Sin esta cláusula no podríamos compilar la clase Operacion al tener parte sin implementar.

```
public abstract class Operacion
{
    protected double n1;
    protected double n2;
    protected double res;
    ...
}
```

Pero, ¿por qué no podríamos compilar la clase Operacion sin la cláusula `abstract` en la definición de la misma? Al contener funciones abstractas, la clase no está completamente definida (está “incompleta”). Siempre que alguna de las funciones de una clase esté definida como abstracta, la clase deberá serlo también para que lo sepa el compilador y nos permita compilar aun estando incompleta (sin implementar todos sus elementos). A la inversa no es necesario, que una clase sea “completa” no impide que sea definida abstracta, sí tenemos algún motivo para hacerlo.

- **Protected**

Utilizamos `protected` para las variables por dos motivos:

- En primer lugar, no nos interesa que sean públicas (`public`), pues no nos interesa que puedan ser consultadas y/o modificadas directamente.

- Segundo, tampoco pueden ser privadas (`private`), debido a que tendremos que utilizarlas desde las diferentes operaciones que vayamos definiendo para realizar el cálculo correspondiente.

La solución es definirlas protegidas (`protected`), privadas hacia fuera y con acceso interno dentro del mismo paquete de clases y todas las subclases.

	Clase	SubClase	Package	Todos
Public	Green	Green	Green	Green
Protected	Green	Green	Green	Red
Private	Green	Red	Red	Red

Figura 2. Visibilidad según sea Public, Protected o Private. En verde se indica que hay visibilidad y en rojo cuando no. Fuente: elaboración propia.

- **Constructor**

Nuestra función constructora tiene dos particularidades:

- En primer lugar, ¿cómo siendo una clase abstracta de la cual no podremos crear objetos, necesitamos un función constructora? Uno de los motivos para crear la clase abstracta `Operacion`, es el de agrupar todo aquello que fuera común a las operaciones. Todas nuestras operaciones tienen en común que tienen que asignar valor a dos variables y calcular el resultado según el tipo de operación al ser creadas. Al definir un constructor para la clase abstracta sólo lo podremos llamar desde las constructoras de las subclases de ésta (`Suma`, `Resta`...).

```
protected Operacion(double num1, double num2)
{
    n1=num1;
    n2=num2;
    res=operar();
}
```

- En segundo lugar, la función constructora invoca el método `operar()`, un método que aún no hemos programado en esta clase. Existirá (estará implementado) solo en las subclases.

```
protected abstract double operar();
public abstract char operador();
```

Aunque no lo implementemos, sí debemos definirlo, y al no estar implementado lo haremos como abstracto (deberemos añadir la cláusula `abstract`). El significado de la cláusula `abstract` en la definición de una función, tiene cierta similitud con el uso para la definición de la clase, con algunas diferencias. Un método abstracto no estará implementado, pero deben hacerlo obligatoriamente las subclases. Así nos aseguramos de que todos los objetos de esta clase tengan esa función y no lo implementamos de una vez porque su comportamiento será diferente en cada una de las subclases. Es evidente que un `operar()` para una suma es diferente al de un resta o una multiplicación.

¿Qué hemos conseguido? Tenemos agrupadas todas las operaciones en una sola clase: podemos crear una lista de operaciones y guardarlas, sin importarnos de qué tipo son. Es más, podemos acceder a su resultado sin preocuparnos por los detalles. Sabemos que todas operaciones, tienen una función `resultado()`, y que esta función devuelve un `double`. Cuando queremos crear nuevas operaciones diferentes, sólo tenemos que copiar, por ejemplo, la clase `Suma` e implementarla modificando las funciones, eliminando aquellas que no usemos y no sean obligatorias y añadir, de ser necesario, aquellas funciones nuevas del nuevo tipo de operación. Todo esto, hace el código más fácil de entender (normalmente) y mantener.

- **Extends**

Un nuevo elemento que tampoco habíamos visto hasta ahora es que al definir la clase añadimos la cláusula `extends` al final seguida por el nombre de una clase. De esta forma indicamos al compilador que la clase actual hereda de la clase indicada al final de su definición:

```
public class Suma extends Operacion
{
    ...
}
```

En este caso particular estamos indicando que `Suma` hereda de `Operacion`. Con el símil de padre/madre hijo/hija, `Suma` es "hija" de `Operacion` y `Operacion` es "madre" de `Suma`.

Hay un elemento importante a destacar en Java, también válido para algunos de los otros lenguajes orientados a objetos, respecto a la herencia: **una subclase puede sustituir una función de su clase "madre" únicamente con volver a definirla como si no existiera previamente** (no hubiese sido heredada). De esta forma, por ejemplo, en `Suma` podríamos volver a definir la función `toString()` si como programadores lo decidíramos, o necesitásemos, debido a que la representación implementada para la transformación a texto en la clase `Operacion` no es la más idónea o eficiente para nuestro caso en particular.

Adicionalmente, de manera implícita en el caso de Java, tenemos la particularidad que todas las clases heredan de la clase `Object`. Si revisamos la documentación de la clase `Object` tal y como se puede ver en la Figura 3:

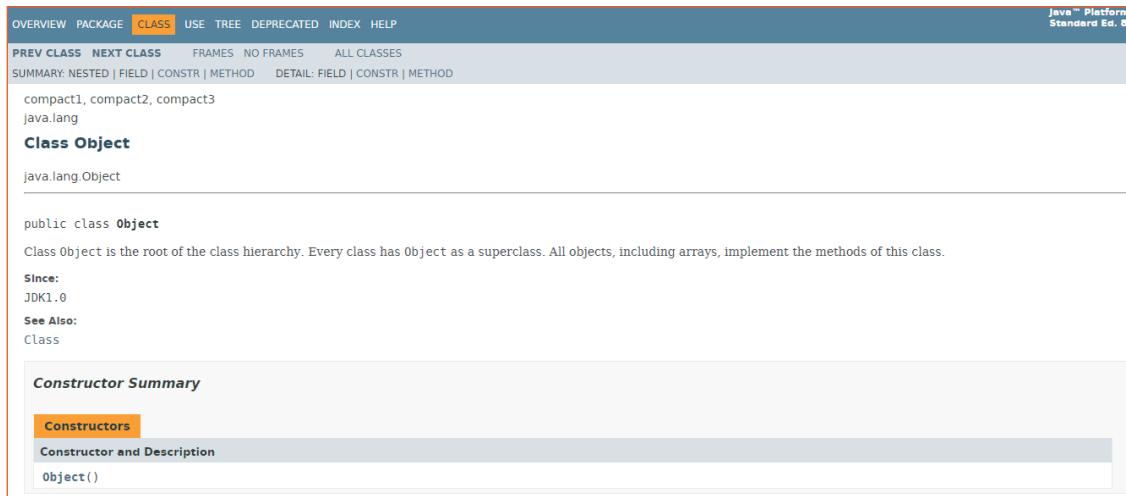


Figura 3. Captura de la documentación de la clase `Object` de la documentación de Java. Recuperado de: <https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

Veremos que dicha clase implementa una serie de funciones que heredan todas las demás clases en Java, entre ellas por ejemplo la función `toString()`. Lamentablemente la conversión que realiza a texto, y por lo tanto el valor devuelto por este método de la clase `Object` es, para decirlo amablemente, no muy bien encontrado, siendo bastante "feo" y poco útil el resultado obtenido por defecto y necesitando en muchos casos que el programador lo sobrescriba (redefina de nuevo la función). Sí que es más útil cuando, por ejemplo, concatenamos `Strings` con otros objetos (instancias de clases) mediante el operador suma `+`, en este caso lo que hace el compilador es traducir el objeto (instancia de la clase) a `String` invocando su método `toString()`. Al heredar todas las clases de `Object` y al tener esta implementada `toString()` seguro que existe y si el programador lo ha redefinido también funcionará.

- **Super**

En la primera línea de la constructora de `Suma` encontramos un llamado a una función que aparentemente no hemos definido:

```
super(n1, n2);
```

Con esta llamada lo que se hace es invocar el constructor de la clase superior-padre-madre. En nuestro caso en particular al constructor de la clase `Operacion`, el cual recibe dos parámetros `double`. En la próxima sección veremos con más detalle las características de los constructores.

1.4 Constructores

En los apartados anteriores hemos visto algún ejemplo, pero para exemplificarlo mejor empecemos con un ejemplo clásico con padre e hija:

Padre

```
public abstract class ClasePadre
{
    public ClasePadre()
    {
        this(null);
        System.out.println("Y no me has pasado parámetros.");
    }
    public ClasePadre(String unparametro)
    {
        System.out.println("Acabas de crear un objeto de la clase padre.");
        if(unparametro != null)
            System.out.println("Me has dicho: "+unparametro);
    }
}
```

Hija

```
public class ClaseHija extends ClasePadre
{
    public ClaseHija()
    {
        System.out.println("Acabas de crear un objeto de la clase hija.");
    }
    public ClaseHija(String unparametro)
    {
        super(unparametro);
        System.out.println("Acabas de crear un objeto de la clase hija.");
    }
    public static void main(String args[ ])
    {
        ClaseHija objeto1=new ClaseHija();
        ClaseHija objeto2=new ClaseHija("Hola");
    }
}
```

Ambas clases tienen la función constructora repetida, con y sin parámetro, eso es así porque Java permite definir una misma función varias veces si en cada definición tiene diferentes parámetros. Por

si no lo habéis intuido las funciones constructoras de una clase son aquellas que tienen el mismo nombre que la clase. En caso de tener múltiples funciones constructoras, por defecto se ejecuta/invoca la que no tiene ningún parámetro.

```
class ClaseHija  
Funciones constructoras → public ClaseHija() y public ClaseHija(String unparametro)
```

Después de ver el código de las clases Padre e Hija y con los ejemplos vistos en los apartados anteriores, ¿os atrevéis a predecir cuál será la salida por pantalla del programa cuando se ejecute? Intentarlo sin avanzar en el manual. Como pista pensad qué debería hacer Java cuando crea un objeto que es subclase de otro. Por ejemplo, en el caso de Operacion y Suma, parece natural que primero será necesario crear la Operacion y después la Suma, de no hacerlo en este orden podríamos dejar sin inicializar (dar valor) una variable de la clase Padre y cuando el Hija la heredara estaría vacía (sin valor, null).

Con la "pista" tenéis una idea aproximada, último intento, en el siguiente párrafo se explica con detalle.

Específicamente el programa producirá la siguiente salida por consola:

1. > Acabas de crear un objeto de la clase padre.
2. > Y no me has pasado parámetros.
3. > Acabas de crear un objeto de la clase hija.
4. > Acabas de crear un objeto de la clase padre.
5. > Me has dicho: Hola
6. > Acabas de crear un objeto de la clase hija.

Vamos a ver paso a paso el porqué de cada una de las líneas mostradas por la consola. La primera línea 1) está codificada en la función constructora de la clase ClasePadre que recibe un parámetro. Recordad que hemos dicho anteriormente que por defecto se ejecuta el constructor que no tiene parámetros, y ¿cómo le decimos que ejecute un constructor diferente al por defecto? Tenemos dos opciones:

- Podemos invocar el constructor que queremos dentro de la misma clase usando `this([parametros])`.
- Podemos invocar un constructor de la clase Madre, usando `super([parametros])`.

Regresando al ejemplo Padre e Hija, en el caso de la clase ClaseHija, no indica nada en su primera línea de la función constructora:

```
public ClaseHija()  
{  
    System.out.println("Acabas de crear un objeto de la clase hija.");  
}
```

Eso hará que Java llame al constructor por defecto de su clase Padre:

```
public ClasePadre()
{
    this(null);
    System.out.println("Y no me has pasado parámetros.");
}
```

Que a su vez llamará al constructor de la clase Padre que tiene un parámetro `this(null)`, ejecutándose:

```
public ClasePadre(String unparametro)
{
    System.out.println("Acabas de crear un objeto de la clase padre.");
    if(unparametro != null)
        System.out.println("Me has dicho: "+unparametro);
}
```

Mostrando el texto "Acabas de crear un objeto de la clase padre." y al ser el parámetro `null` no mostrando el texto "Me has dicho" sino directamente el texto "Y no me has pasado parámetros" de la constructora del `ClasePadre` sin parámetros y posteriormente el texto de la `ClaseHija` "Acabas de crear un objeto de la clase hija."

Adicionalmente para aclarar el comportamiento por defecto de Java en las funciones constructoras, en el caso que estas no invoquen ningún otro constructor desde la función constructora de una clase, es igual que si hubiésemos puesto en la primera línea de la función:

```
super()
```

Con lo que se invoca el constructor por defecto de nuestra clase `ClasePadre`, el que no recibe ningún parámetro.

1.5 Interfaces

Algunos lenguajes orientados a objetos permiten herencia múltiple, heredar de más de una clase. En el caso de Java sólo podemos heredar de una única clase Padre/Madre. Normalmente con la herencia simple (de una única clase) será suficiente para nuestro programas, siempre que cuidemos cómo estructuramos el problema. De todas formas, en los casos en los que no consigamos estructurarla de forma de limitarnos a herencia simple y necesitemos "implementar" una herencia múltiple, Java nos da un camino alterno para hacerlo usando las interfaces.

Una *Interface*, no es más que una clase **completamente abstracta**, es decir, una clase en la cual todas sus funciones son abstractas (no están implementadas), y todos sus atributos son constantes. Para declarar un atributo como constante usaremos `final`, por ejemplo:

```
public final String NOMBRE="Roger";
```

Con el uso de las interfaces, Java permite superar la limitación de una sola herencia, pudiendo nuestras clases heredar de una clase Madre y de tantos interfaces como necesitemos. Permitiendo establecer múltiples Madres o Padres para nuestras clase, indicando que funciones debe tener y como está relacionada con otras clases.

Veamos un ejemplo de código usando tres clases en el que queremos definir una clase de objetos que deben tener una función para obtener el nombre `obtenerNombre()`:

InterfaceNombre

```
public interface InterfaceNombre
{
    public final String NOMBRE =“Roger”;
    public abstract String obtenerNombre();
    public abstract void asignarNombre(String nombre);
}
```

ClaseSimple

```
public class ClaseSimple
{
    public ClaseSimple()
    {
        System.out.println(“Una clase cualquiera.”);
    }
}
```

ClaseHerenciaMultiple

```
public class ClaseHerenciaMultiple extends ClaseSimple implements InterfaceNombre
{
    String nombre;
    public ClaseHerenciaMultiple()
    {
        nombre="";
    }
    public String obtenerNombre()
    {
        return nombre;
    }
    public void asignarNombre(String nombre)
    {
        this.nombre=nombre;
    }
    public static void main(String args[ ])
    {
```

```

InterfaceNombre claseconnombre;
claseconnombre=new ClaseHerenciaMultiple();
claseconnombre.asignarNombre(InterfaceNombre.NOMBRE);
System.out.println(claseconnombre.obtenerNombre());
}
}

```

Diferencias y puntos relevantes respecto a los ejemplos visto hasta ahora, en la interface:

```

public interface InterfaceNombre
{
    public final String NOMBRE = "Roger";
    public abstract String obtenerNombre();
    public abstract void asignarNombre(String nombre);
}

```

Es muy parecida a la declaración de una clase "normal", simplemente que en este caso lo declaramos como **interface**. Adicionalmente cualquier clase que implemente InterfaceNombre debe tener implementadas si o si las funciones **obtenerNombre()** y **asignarNombre()**, no hay ninguna limitación a cómo lo harán, pero no será posible dejarlas sin implementar.

Adicionalmente, hay una variable declarada que usa la cláusula **final**, con lo cual será una "variable" constante. Una vez tenga el valor asignado al inicializarse la clase, este valor no podrá modificarse durante el resto de la ejecución del programa. Las "variables" constantes no sólo pueden declararse en interfaces, sino en cualquier clase que puedan ser de utilidad.

final no sólo sirve para declarar "variables" constantes, también podemos declarar funciones constantes. En ese caso lo que estamos limitando, es que las subclases redefinan la función. Y rizando el rizo, podemos hasta declarar una clase como **final**, en este caso el resultado es que dicha clase no podrá ser heredada, en otra palabras no podrá ser Padre o Madre de otra clase.

Para finalizar, notar que en el caso de las interfaces no se usa la cláusula **extends** para "heredarlas" sino que se usa la cláusula **implements**. Si queremos implementar más de una interface, simplemente lo indicamos separándolas por comas en la declaración.

Hay otro detalle en el **main**, la declaración de la variable **claseconnombre**. Podríamos tener la declaración de tres formas diferentes y las tres serían validas:

```

ClaseHerenciaMultiple claseconnombre= new ClaseHerenciaMultiple();
ClaseSimple claseconnombre= new ClaseHerenciaMultiple();
InterfaceNombre claseconnombre=new ClaseHerenciaMultiple();

```

La primera es lo que normalmente hacemos, para las otras dos hace falta introducir el polimorfismo como veremos en el siguiente apartado.

1.6 Polimorfismo

Cuando dentro de la programación orientada a objetos hablamos de polimorfismo nos referimos a la **característica que hace posible utilizar llamadas iguales (sintácticamente) a objetos de tipos distintos, siempre que estos tengan implementada una respuesta para dicha llamada.**

Otra forma de verlo es que implementaciones diferentes, asociadas a diferentes clases, pueden compartir el mismo método para llamarlas. Según la clase se utiliza su función/implementación específica.

Todo eso puede sonar muy complicado, pero veámoslo con unos ejemplos. Si una Suma hereda de Operacion, "es" una Operacion, no deberíamos tener problema para guardar una variable de la clase Suma en una variable de la clase Operacion, al fin y al cabo, Suma hereda todos los elementos/características de Operacion. La misma situación se produce con ClaseSimple, InterfaceNombre y ClaseHerenciaMultiple, cualquier variable del tipo ClaseHerenciaMultiple es "al mismo tiempo" una variable del tipo ClaseSimple y/o InterfaceNombre.

Hay que ir con cuidado fijándose en quiénes son las clases Padre/Madre, pues las clases Hija/Hijo si "contienen" a los padres (heredan) pero a la inversa no es correcto. Podríamos intentar forzarlo usando en el caso de Operacion, Suma y Resta:

```
Operacion operacion=new Suma();
Suma suma=(Suma) operacion;
Resta resta=(Resta) operacion;
```

En un principio parecería funcionar y conseguiríamos que Java lo compilase, pero una vez en ejecución, cuando tratáramos de realizar la asignación, funcionaría para la primera (pues hemos creado operación como Suma), pero no así en la segunda que tenemos una Resta, lanzando una excepción.

El polimorfismo es la posibilidad de realizar esta asignación a una clase que no es exactamente la del objeto en cuestión, sino una de sus Padre/Madre (pudiendo recuperar el objeto en su tipo original). Dependiendo de la clase de la variable que tengamos asignado el objeto, puede "hacerse pasar" por una clase u otra, pero el objeto siempre será el mismo. Esto nos sirve, por ejemplo, para guardar una lista de operaciones en el caso de la calculadora, que de hecho no serán Resta, Suma, Multiplicacion, etc... (todas ellas extienden/heredan Operacion).

1.7 Elementos estáticos: atributos y funciones

En los ejemplos hemos utilizado algunos atributos (de la clase System, por ejemplo), sin instanciar/crear ningún objeto de la clase. ¿Cómo ha podido funcionar? esto es posible gracias a que podemos definir funciones y atributos como estáticos usando la cláusula static. **Una variable o función definida como estática “pertenece” a todos los objetos de la clase y, podemos referenciarlo sobre el mismo nombre de la clase** (por ejemplo, System.in).

No se recomienda usar elementos estáticos, a no ser que sea muy justificado. Por ejemplo, sirve para hacer que todos los objetos de una clase compartan el mismo atributo o que podamos llamar una función de una clase sin necesidad de instanciarla. Un ejemplo típico de uso es la clase Math, la cual tiene todos sus funciones estáticas. Podríamos decir que Math funciona más como una librería de utilidades que una clase propiamente dicha.

1.8 Captura de excepciones

Las excepciones **son la manera que tienen los programas de informarnos de errores producidos durante la ejecución**. Adicionalmente, el hecho de generar/lanzar excepciones es una manera de asegurarnos que el programador trata dichos errores, o como mínimo es consciente que se pueden producir y sino los trata deberá lanzarlos a su vez para que sean atendidos en un nivel superior (usando throws).

Cuando una función pueda producir/lanzar una excepción, será necesario capturarla usando la cláusula try (enmarcando todo el código que puede producir exceptions) y tratarla usando las cláusulas catch (será necesario indicar específicamente qué excepción o excepciones estamos "capturando") y finally (acciones a ejecutar tanto si se han producido excepciones en el bloque delimitando por try como si no). Veámoslo con un ejemplo:

LeerTeclado

```
import java.io.*;
public class LeerTeclado
{
    public static void main(String args[])
    {
        InputStreamReader reader=new InputStreamReader(System.in);
        BufferedReader entrada=new BufferedReader(reader);
        try
        {
            System.out.println("Voy a leer una línea de teclado.");
            entrada.readLine();
        }
        catch (IOException x)
        {
            System.out.println("Error al leer");
        }
        finally
        {
            System.out.println("Fin, tanto si la lectura fue correcta como sino");
        }
    }
}
```

El código no tiene mucha dificultad y es muy parecido a otros lenguajes, por ejemplo a Python, que usamos en asignaturas anteriores. Sólo una "curiosidad", ¿qué es la x que aparece dentro de la cláusula catch? En Java, una excepción, como casi cualquier elemento en los programas (incluso los errores) son objetos, exceptuando los tipos primitivos. De esta manera cuando capturamos una excepción, la guardaremos en una variable para poder tratarla.

Y, ¿cómo podemos tratarla? Por ejemplo, usando la función getMessage(), que tienen todos los errores, con lo que conseguiremos una descripción de la excepción en forma de String.

El ejemplo de lectura de teclado será difícil de probar, pues sólo se producirá una excepción si la lectura de la entrada estándar (normalmente el teclado), nos produce una excepción de entrada / salida.

1.9 Lanzamiento de excepciones

Las excepciones se pueden producir por errores en el programa o también podemos crearlas nosotros como programadores. Para lanzar una excepción usaremos la cláusula throw, ya sea lanzando la excepciones predefinidas en Java o hasta podemos crear nuestras propias excepciones creando una clase que herede de Exception. Veamos un ejemplo:

Lanzar

```
public class Lanzar
{
    public void lanzar(boolean lanzar) throws Exception
    {
        if(lanzar)
        {
            Exception exc;
            exc=new Exception("Mi excepción.");
            throw exc;
        }
    }
    public static void main(String args[]) throws Exception
    {
        Lanzador lanzador=new Lanzador();
        lanzador.lanzar(false);
        System.out.println("Ejecutándose.");
        lanzador.lanzar(true);
        System.out.println("Ya no estoy ejecutándome.");
    }
}
```

El programa mostrará por consola:

```
> Ejecutándose.  
> java.lang.Exception: Mi Excepción.  
  at Lanzar.lanzar(Lanzar.java:8)  
  at Lanzar.lanzar(Lanzar.java:20)
```

La excepción interrumpe la normal ejecución del programa, provocando que el `main` termine y sin llegar a ejecutarse la última línea de código. Si modificamos el programa incluyendo `try` y `catch` podemos controlar como reacciona delante de la excepción y no terminará abruptamente:

LanzaCaptura

```
public class LanzaCaptura  
{  
    public static void main(String args[])  
    {  
        Lanzador lanzador=new Lanzador();  
        try  
        {  
            lanzador.lanzar(false);  
        }  
        catch(Exception x)  
        {  
            System.out.println("Excepción: "+x.getMessage());  
        }  
        System.out.println("Ejecutándose.");  
        try  
        {  
            lanzador.lanzar(true);  
            System.out.println("Ya no estoy ejecutándome.");  
        }  
        catch(Exception x)  
        {  
            System.out.println("Excepción: "+x.getMessage());  
        }  
        System.out.println("Ejecutándose 2.");  
    }  
}
```

Este ejemplo se ejecutará completamente sin interrumpirse abruptamente. También vemos que la cláusula `throws` en el `main` ha desaparecido, pues al capturar la excepción ya no es necesario pasar la responsabilidad del tratamiento a otro.

Java no permite ignorar las excepciones, deberemos tratarlas (catch) o lanzarlas (throws) para que otro las trate. Dependiendo del programa y de cada situación particular, el programador deberá decidir qué hacer.

1.10 Errores

La clase Error es una subclase de Throwable, podríamos definir los errores como **excepciones graves que por norma general deben abortar la ejecución del programa**. Por ejemplo, un error de la máquina virtual de Java o quedarnos sin memoria.

Aun siendo muy graves nada nos impide capturarlos e intentar tratarlos para recuperarnos. Pero por lo general eso no será posible y terminaremos teniendo que abortar de todas formas.

Tema 2. Especificación: diagramas de clases y casos de uso

Para formalizar la especificación de los requisitos funcionales de un sistema, unas de las herramientas más utilizadas son los diagramas de clases y los casos de uso. En ambos casos usaremos los diagramas definidos dentro del estándar UML, el más usado a nivel mundial tanto en la academia como en la industria.

2.1. UML

El **Unified Modeling Language** (UML), está definido y mantenido como estándar por el *Object Management Group* (OMG). El OMG es un consorcio sin ánimo de lucro internacional que promueve estándares para trabajar con programación orientada a objetos, fundado en 1989. Entre los miembros de OMG se encuentran los principales actores del sector, como por ejemplo Microsoft e IBM entre muchos otros. Otros estándares desarrollados por OMG son, por ejemplo: XMI, CORBA o BPMN (López, Costa y Samsó, 2004).



UML

<https://www.uml.org/>



OMG

<https://www.omg.org/>

UML se ayuda de diferentes representaciones gráficas, o diagramas del sistema para definir los requerimientos que este deberá cumplir. Aun cuando hay más de diez tipos diferentes de diagramas, nosotros nos concentraremos en sólo dos de ellos en este curso, los diagramas de clases y los casos de uso. En otras asignaturas del grado se explicarán diagramas adicionales como pueden ser, por ejemplo: los diagramas de secuencia o los diagramas de actividad (Kimmel, 2007).

2.1.1 Diagramas de clases

Los diagramas de clases se combinan especialmente bien cuando trabajamos con lenguajes orientados a objetos, ya que estos sirven como esqueletos para la implementación de los objetos de nuestro sistema. En la Figura 4 podemos ver un ejemplo de un diagrama de clases.

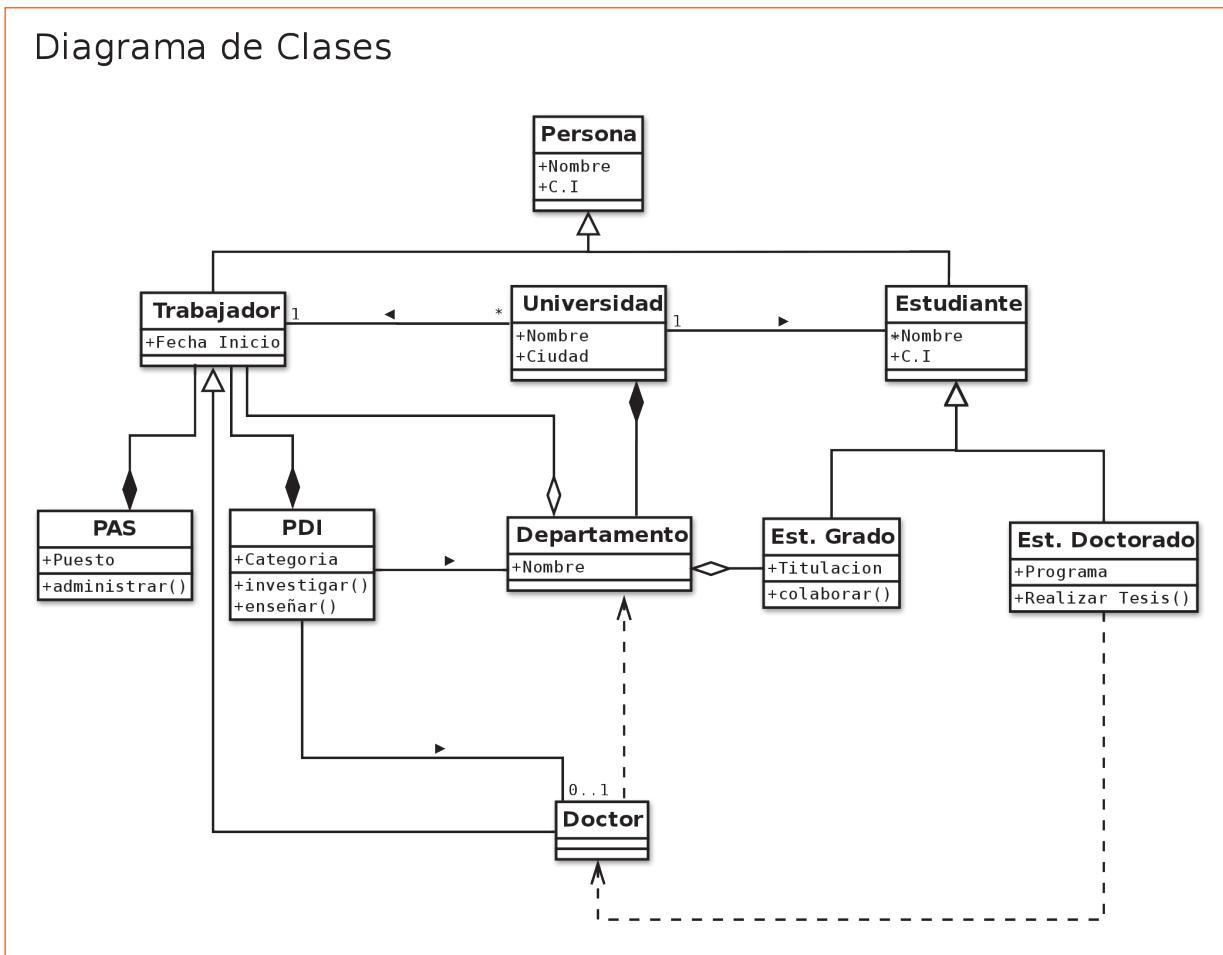


Figura 4. Ejemplo de diagrama de clases. Recuperado de: Creative Commons Attribution-Share Alike 3.0 Unported, The Photographer.

Los diagramas de clases permiten expresar gráficamente sus diferentes componentes:

- Atributos
- Funciones
- Visibilidad

Así como las relaciones entre las diferentes clases:

- Herencia
- Composición
- Agregación
- Asociación
- Uso

La representación de un clase se divide en tres bloques: nombre, atributos y sus funciones, tal y como se puede ver en la Figura 5.



Figura 5. Representación gráfica de una clase con sus tres bloques: nombre, atributos y funciones. Fuente: elaboración propia.

Si queremos indicar que una clase es abstracta, deberemos poner el nombre de la clase en *cursiva*.

Para representar la visibilidad de los atributos y funciones, se indicará delante del nombre de cada uno de ellos el siguiente símbolo:

- → *private*: únicamente visible por él.

→ *protected*: visible por él y sus hijos.

+ → *public*: visible para todos.

Por ejemplo, en la Figura 4, el nombre de las personas (clase persona) es público (+Nombre) o la función investigar() de los PDI también es pública (+investigar()).

Para representar las cardinalidad de las relaciones entre clases se usan las siguientes notaciones, separadas por / las diferentes opciones con igual significado:

- Uno o muchos: 1..* / (1..n) / (-)
- Cero o muchos: 0..* / (0..n)
- Número fijo: m

Para indicar que una clase hereda de otra se utiliza una flecha, la clase Hija apunta a la clase Madre. Por ejemplo, Auto hereda de Vehículo como se muestra en la Figura 6.

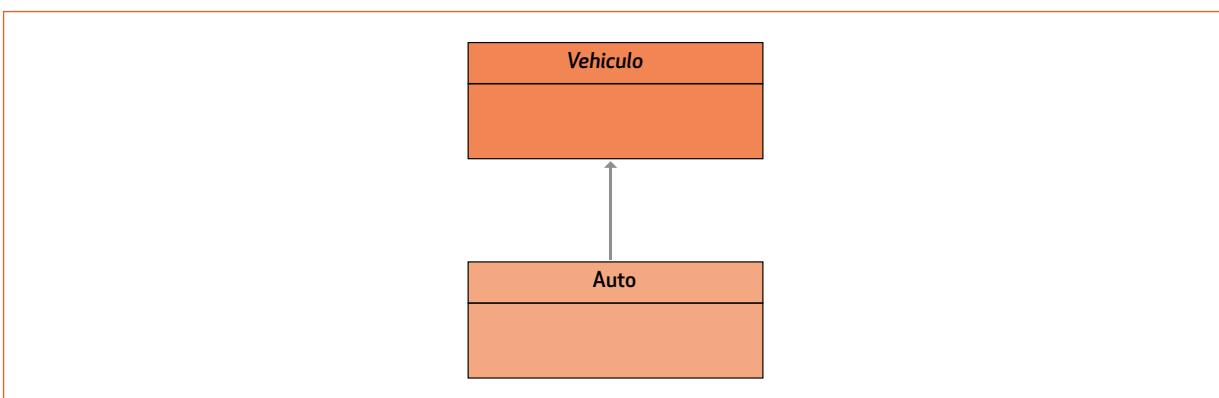


Figura 6. Representación gráfica de la herencia entre dos clase, Padre = Vehículo, Hijo = Auto. Fuente: elaboración propia.

También podemos representar las relaciones de agregación donde tenemos dos casos:

- **Composición (por valor)**

Relación estática, el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye.

- **Agregación (por referencia)**

Relación dinámica, tiempo de vida del objeto incluido independiente del que lo incluye.

La Figura 7 muestra las dos opciones de agregación:

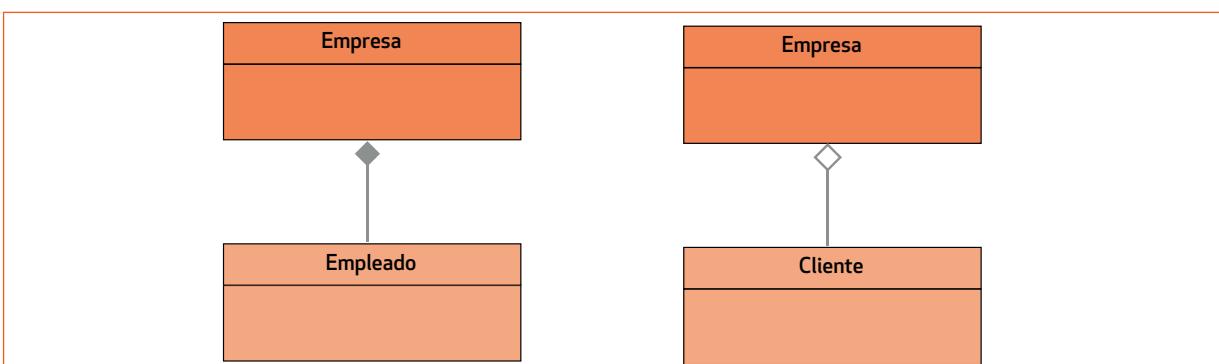


Figura 7. Agregación por valor y por referencia. Composición Empresa ← Empleado, Agregación Empresa ← Cliente. Fuente: Elaboración propia.

En internet se puede encontrar multitud de ejemplos de diagramas de clases, por ejemplo, en el siguiente enlace:

<https://sites.google.com/site/wwwprogramacionorientadacom/12-lenguaje-unificado---umc-diagrama-de-clases>

2.1.2 Casos de uso

Los casos de uso representan la relación entre los diferentes actores (*stakeholders*) y los casos de uso del sistema. Muestran la funcionalidad que ofrece el sistema en su interacción externa. Describen qué hace el sistema, pero no cómo lo hace.

Son útiles para definir cómo debe comportarse el sistema. Utilizados principalmente como parte de la documentación del código, facilitando la reutilización del mismo.

Los diagramas de casos de uso se componen de los actores, representados por el gráfico esquemático de una persona, y los casos de uso, representados por globos con un breve texto que identifica el caso. Adicionalmente podemos agrupar los diferentes casos que componen un elemento del sistema utilizando un cuadrado o rectángulo.

Para finalizar podemos tener diferentes relaciones entre los diferentes casos:

- **Inclusión (include)**

Un caso de uso que forma parte de otro caso de uso.

- **Extensión (extends)**

Funcionalidad adicional opcional.

- **Generalización**

Equivalente a herencia.

En la Figura 8 podemos observar las diferentes notaciones para los diagramas de casos de uso y en la Figura 9 un ejemplo de caso de uso para un restaurante.

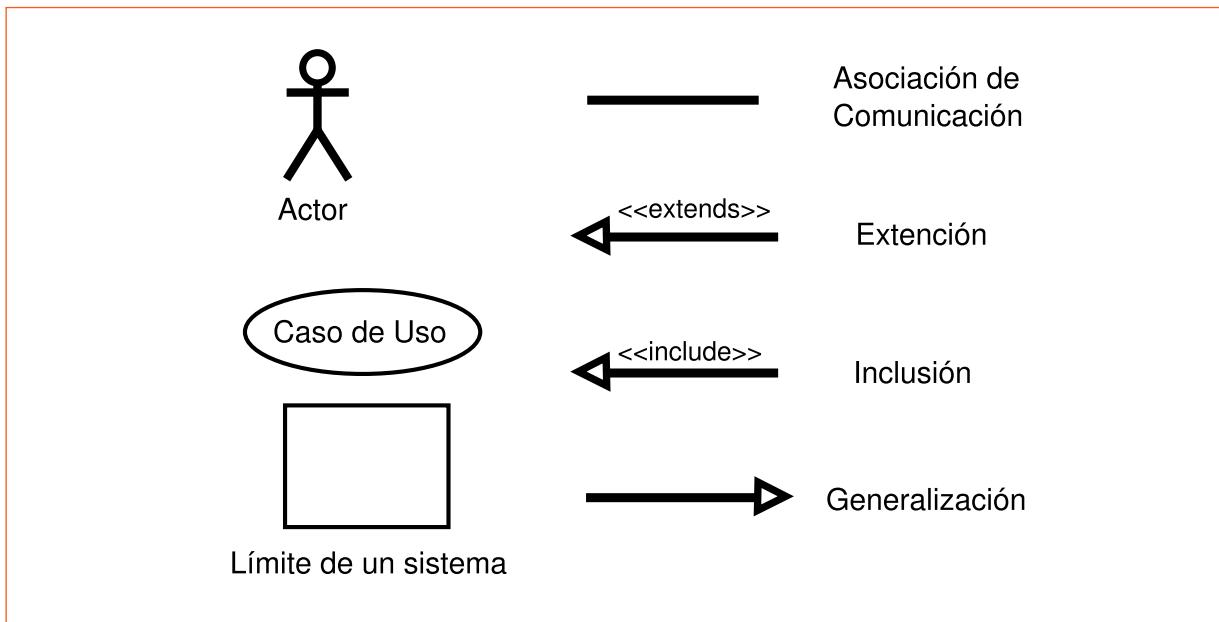


Figura 8. Notación casos de uso. Recuperado de: Creative Commons Attribution-Share Alike 3.0 Unported, The Photographer.

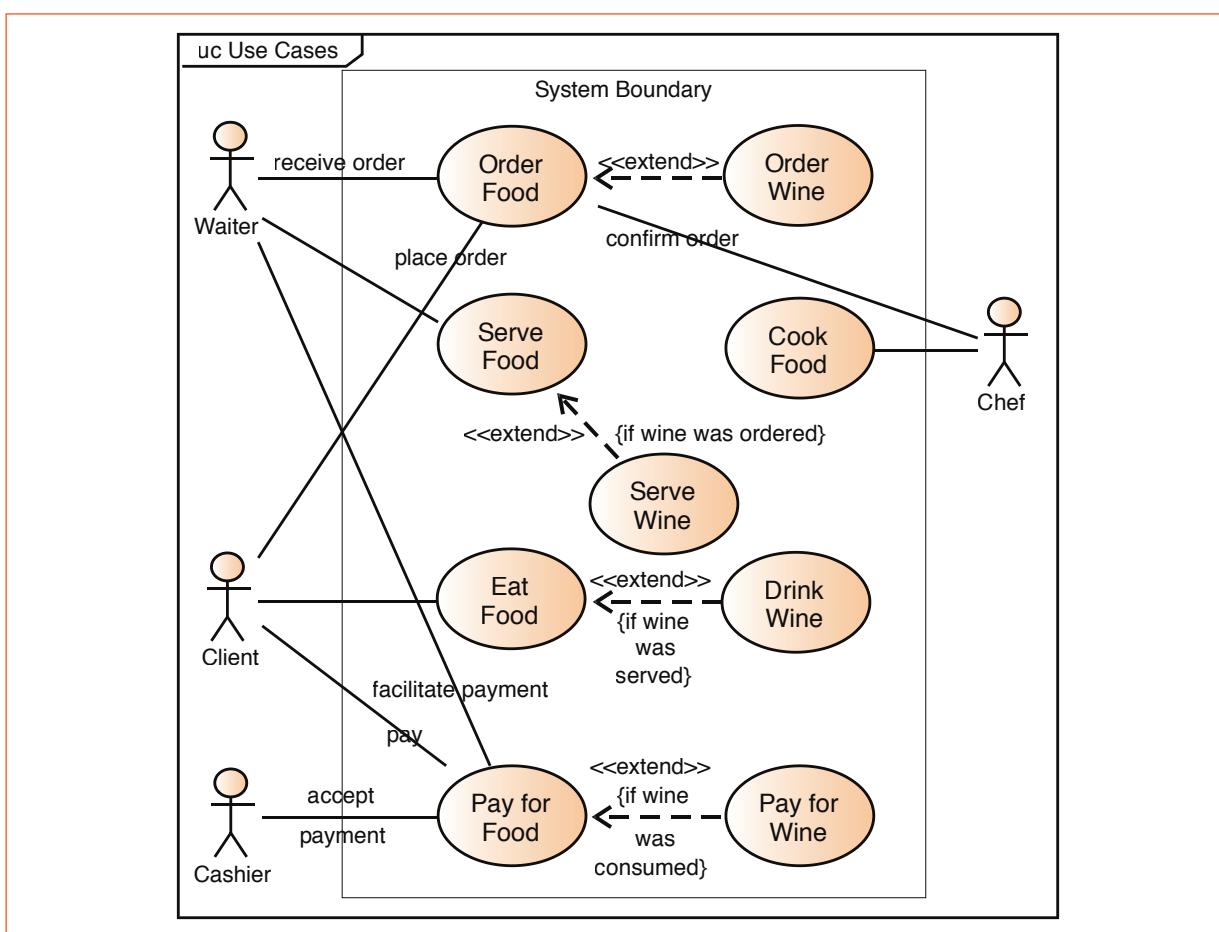


Figura 9. Ejemplo de caso de uso restaurante. Recuperado de: Creative Commons Attribution-Share Alike 3.0 Unported, Kishorekumar 62.

Tema 3. Diseño: arquitectura en 3 capas

La programación por capas es un modelo de desarrollo software en el que el objetivo primordial es la independencia (abstracción / separación / desacoplamiento) de las partes que componen un sistema.

Las ventajas de la arquitectura en capas son:

- Mayor escalabilidad.
- Facilitación del mantenimiento.
- Incremento de las posibilidades de reutilización del código.

En contra partida tenemos:

- Reducción del redimiendo debido a la necesidad de paso de mensajes entre capas.
- Mayor dificultad inicial en la programación debido a que debemos realizar correctamente la granularidad de las diferentes capas para evitar problemas durante la implementación de cada una de ellas.

Las ventajas son normalmente mayores a los inconvenientes y es una de las arquitecturas más usadas en la actualidad en múltiples contextos.

Normalmente se utilizan 3 capas a las cuales se les asigna las siguientes responsabilidades:

- **Presentación** (Visualización)

Capa que interactúa directamente con el usuario. Presenta el sistema al usuario, le comunica la información, captura las acciones realizadas por él y la información introducida con un mínimo de proceso (por ejemplo, puede realizar un filtrado básico de los datos para comprobar que no hay errores de formato, por ejemplo, que la fecha sea correcta y dentro del rango esperado en las edades). Debe proporcionar una experiencia lo más ergonómica posible al usuario, facilitándole el trabajo. Esta capa se comunica únicamente con la capa inmediatamente inferior, la de funcionalidad.

- **Funcionalidad** (Lógica de negocios)

Capa donde se realizan todos los cálculos necesarios para dar respuesta a las peticiones del usuario siguiendo la lógica del negocio. Recibe las peticiones del usuario, de ser necesario se comunica con la capa de persistencia para obtener datos, realiza el proceso y envía la respuesta al usuario. Esta capa se comunica con las dos otras capas, tanto la de presentación como la de persistencia.

- **Persistencia** (Datos)

Capa encargada de almacenar los datos y gestionar el acceso a los mismos. Normalmente incluye un gestor de bases de datos como elemento principal. Sólo recibe solicitudes de almacenamiento o recuperación de información desde la capa de funcionalidad.

Un ejemplo clásico podría ser el que se observa en la Figura 10, donde tenemos varios clientes con equipos de escritorio, un servidor de negocio y un servidor de bases de datos.



Figura 10. Ejemplo de arquitectura en 3 capas. Recuperado de: https://es.wikipedia.org/wiki/Programaci%C3%B3n_por_capas.

En nuestro caso en particular, dado que estamos trabajando en esta asignatura con Java, podemos tener que las 3 capas pueden estar implementadas puramente con Java, como se ve en el ejemplo de

la Figura 11. Hasta podríamos ser un poco más específicos y también añadir una Bases de Datos, por ejemplo, podemos implementar la presentación usando Java AWT, la funcionalidad usando Java para por ejemplo un programa que administre mesas electorales y para la persistencia podemos combinar Java con MySQL como se muestra en la Figura 12.



Figura 11. Ejemplo de arquitectura en 3 capas puramente en Java. Fuente: elaboración propia.



Figura 12. Ejemplo de arquitectura en 3 capas combinado un *package* específico de Java (AWT), Java y MySQL. Fuente: elaboración propia.

Una variante con las mismas capas, pero con diferente nombre es el patrón **Modelo Vista Controlador** (MVC). Donde tenemos que el modelo se responsabiliza de los datos, el controlador de la funcionalidad y la vista de la presentación.

Inicialmente fue desarrollado para aplicaciones de escritorio. Posteriormente se adaptó y sigue siendo muy utilizada para aplicaciones web, donde tenemos una división entre cliente (Vista) – servidor (Modelo). El controlador puede estar implementado en ambos lados o más en uno que en el otro. En la Figura 13 podemos ver el proceso de interacción del usuario con el MVC.

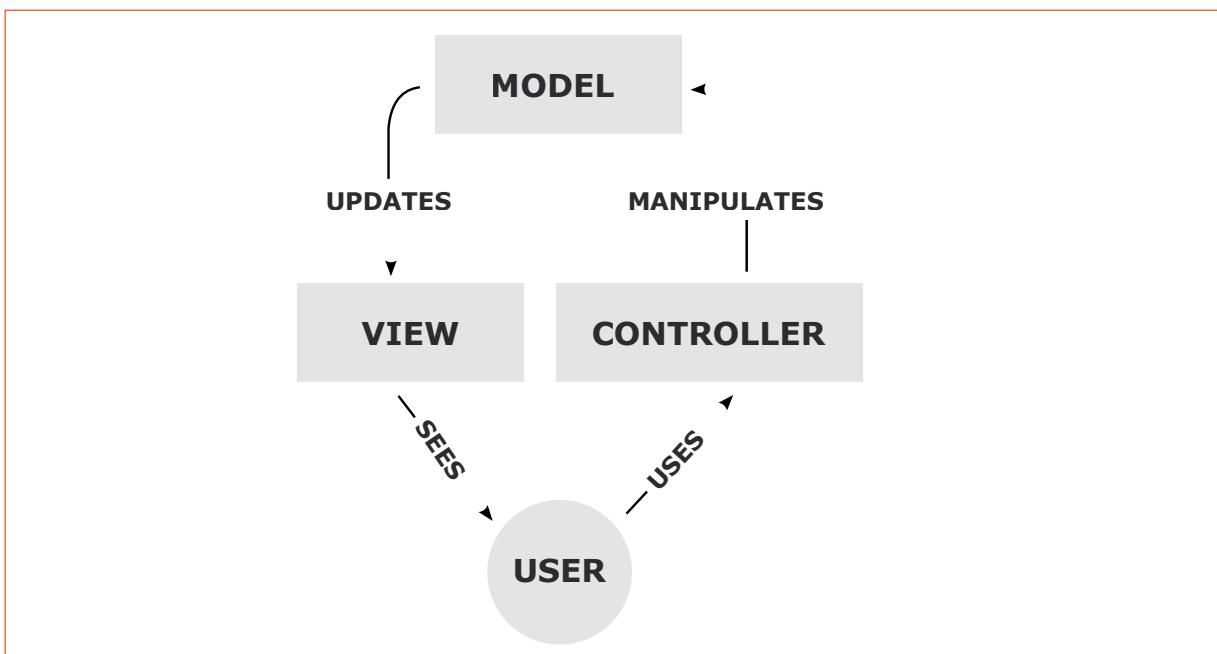


Figura 13. Modelo Vista Controlador. Recuperado de: <http://w3programmers.com/page/2/>

Tema 4. El lenguaje de programación Java

Java es un lenguaje de programación Orientado a Objetos (OO) multipropósito, en otras palabras, capaz de implementar soluciones para casi cualquier problema.

Cuando fue creado se estableció como prioridad que fuese fácilmente portable entre diferentes plataformas, al mismo tiempo considerando el general bajo rendimiento de los lenguajes interpretados (que es una de las maneras más comunes de conseguir esta portabilidad) se llegó a un compromiso. Es un lenguaje interpretado, por lo que normalmente, se genera en primer lugar un *bytecode* y es este el que es portable al poder ejecutarse en las diferentes máquinas virtuales java disponibles para cada plataforma. Pero tenemos también la posibilidad de generar código máquina para mejorar la velocidad de ejecución a costa de perder la portabilidad.

Fue creado por un equipo liderado por James Gosling en Sun Microsystems y su primera versión publica fue lanzada en 1995. En el año 2010 Oracle Corporation adquirió Sun Microsystems, y ha seguido con el desarrollo de Java hasta la fecha. Tienen una similitud con C y C++, pero no llega a tener la capacidad de estos para trabajar a bajo nivel.



Figura 14. Logo de Java. Recuperado de: Oracle Corporation, <https://www.java.com/es/about/>.

4.1. Máquina Virtual Java

Vamos a entrar en más detalle en cómo consigue Java la portabilidad. Hablábamos en el punto anterior de que el código en Java se compila en un *bytecode* en vez de en el lenguaje máquina nativo de donde queremos ejecutar nuestro programa. Lo que hace es generar el código máquina de un procesador que no existe, que es la **Máquina Virtual Java** (*Java Virtual Machine*, JVM). Esta máquina virtual es una emulación completa con procesador, sistema operativo, etc. y se tiene que implementar para cada plataforma que dé soporte a Java. Actualmente, existe para Windows, Unix, Solaris y la mayoría de sistemas operativos. Podemos hacer un símil con los emuladores que nos permiten ejecutar el código de otro ordenador en nuestro PC. En este caso, el ordenador que estamos emulando es a su tiempo un artificio para disponer de portabilidad.

Como programadores podemos obviar los detalles de la Máquina Virtual Java, pues para programar nos será totalmente transparente. Solo deberemos preocuparnos en el despliegue de nuestro *software* que el equipo tenga instalada su correspondiente JVM para poderlo ejecutar sin contratiempos.

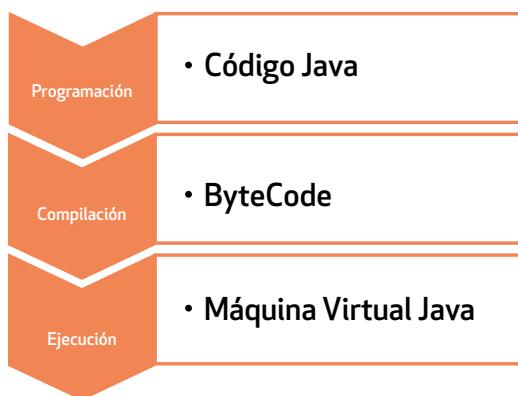


Figura 15. Código - bytecode - JVM. Fuente: elaboración propia.

4.2. Introducción al lenguaje

Dado que en las diferentes asignaturas anteriores del grado ya hemos visto cómo programar desde cero, empezaremos con el típico ejemplo de "Hola Mundo" para introducir el lenguaje Java.

Ejemplo Hola Mundo

```
import java.lang.*;  
public class HolaMundo  
{  
    public static void main(String args[ ])  
    {  
        System.out.println("¡Hola Mundo!");  
    }  
}
```

Hay cosas familiares con otros lenguajes y algunas nuevas. Veamos línea por línea:

```
import java.lang.*;  
...
```

Con esta línea indicamos de manera similar a, por ejemplo, en Python, que vamos a utilizar elementos externos. Si en Python sería en modulo, en Java al ser orientado a objetos, el `import` indica que utilizaremos alguna (con el `*` denotamos que puede ser cualquiera de las librerías contenidas en `java.lang`) en nuestro programa. Específicamente, usaremos la librería `System`. Dos particularidades: en primer lugar si solo vamos a utilizar una librería podríamos importar únicamente esta usando `import java.lang.System;` y en segundo lugar la librería `System` es especial, pues es importada de forma automática aunque no lo especifiquemos.

```
import java.lang.*;  
public class HolaMundo  
{  
    ...  
}
```

Para definir la clase `HolaMundo` utilizamos `public class HolaMundo`. Usamos `public` para indicar que será una clase pública, más adelante veremos las diferentes opciones de visibilidad que tenemos para las clases. La palabra reservada `class` indica que estamos definiendo una clase y el `HolaMundo` es el nombre que le estamos dando. Como en todos los lenguajes se recomienda que el nombre esté relacionado con el contexto de lo que estamos programando. `{ ... }` Las dos llaves delimitan el inicio y fin de la clase `HolaMundo`.

```
import java.lang.*;  
public class HolaMundo  
{  
    public static void main (String args[ ])  
    {  
        ...  
    }  
}
```

En Java debemos definir la función inicial que será llamada cuando ejecutemos la clase. Esta función inicial, como en otros lenguajes, recibe el nombre de `main` y así deberá llamarse obligatoriamente. A las demás funciones seremos libre de ponerles el nombre que nos parezca más adecuado cumpliendo algunas restricciones que veremos más adelante.

Es importante destacar que en caso de que la clase nunca vaya a ser ejecutada no será necesario definir dicha función.

`public` nos indica que la función es pública, de forma parecida a cómo se define para las clases.

`static` nos indica que la función es estática, más adelante veremos otras opciones.

Ambos parámetros, `public` y `static`, son necesarios para cualquier función `main`.

`void` nos indica que la función regresa "nada" (`null`), caso en el que entraremos con más detalle posteriormente. En otros lenguajes no es necesario indicarlo cuando estamos en este caso de regresar "nada" o por ejemplo en C la función `main` regresa `int`.

Como parámetro de entrada recibe siempre un vector de `Strings`, `String args[]`. En Java los vectores tienen asociada su longitud, así que de necesitar saber cuántos parámetros de entrada hemos recibido lo podremos consultar utilizando `args.length`.

```
import java.lang.*;
public class HolaMundo
{
    public static void main(String args[ ])
    {
        System.out.println("¡Hola Mundo!");
    }
}
```

Las primeras partes eran más simples, aquí tenemos la línea de código más compleja para nuestro primer programa: `System.out.println("¡Hola Mundo!");`; vayamos por cada uno de sus elementos por separado.

`System` es una clase que contiene, entre otras cosas, una variable pública llamada `out`, del tipo (clase) `PrintStream`. Los objetos de esta clase tienen una función llamada `println`, que imprime el texto (`String`) que le pasemos seguido por un salto de línea, en este caso le pasamos como parámetro el `String` que queremos que nos imprima en este caso “¡Hola Mundo!”.

El ; como en el caso del `import` es el indicador de final de línea de código.

Entrando más en detalle, la variable `out` de la clase `System` representa la salida estándar del programa, normalmente (por defecto) en la pantalla, pero podemos redireccionarla de ser necesario.

De la misma forma, la clase `System` contiene una variable del mismo tipo llamada `err` (salida de error) que también normalmente está redireccionada a la pantalla, y una variable del tipo `InputStream`

llamada `in`, que corresponde a la entrada estándar (normalmente el teclado). Evidentemente, los objetos de la clase `InputStream` no tienen ninguna función del estilo `println`, pues son para recibir datos, no para sacarlos.

Para acabar de entender la sintaxis, veamos otro ejemplo: el programa “¿Cómo me llamo?” pregunta al usuario su nombre y se lo escribe en pantalla. Es también un ejemplo bastante sencillo, pero empieza a utilizar características particulares del lenguaje Java que interesa que se entiendan correctamente para poder avanzar a cosas más complejas. Si al principio nos cuesta de entenderlo aún con las explicaciones, no hay que preocuparse, pueden seguir leyendo y una vez vistos los otros conceptos regresar.

Ejemplo ¿Cómo me llamo?

```
import java.lang.*;  
import java.io.*;  
public class ComoMeLlamo  
{  
    public static void main(String args[ ]) throws IOException  
    {  
        InputStreamReader reader;  
        BufferedReader entrada;  
        String nombre;  
        reader=new InputStreamReader(System.in);  
        entrada=new BufferedReader(reader);  
        System.out.print("¿Cómo te llamas? ");  
        nombre=entrada.readLine();  
        System.out.println("Te llamas "+nombre);  
    }  
}
```

Respecto al primer ejemplo de `HolaMundo` vemos que necesitamos que el programa nos haga una pregunta, eso podemos hacerlo de manera análoga al “¡Hola Mundo!”. Lo siguiente que debemos hacer es leer el nombre por teclado que nos introducirá el usuario. En una primera aproximación rápida se nos podría ocurrir utilizar `readline()`:

```
nombre=System.in.readLine();
```

Lamentablemente, no es tan fácil como parece. Debido a la clase a la que pertenece la variable `in` de la clase `System` se trata de una variable del tipo `InputStream` y, si miramos la documentación:

<https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>

Vemos que contiene funciones para leer tipos primitivos, pero para nada parecido a un `String`, que es un conjunto de caracteres. Podríamos utilizarla, pero tendríamos que leer byte a byte cada uno de los caracteres que componen el nombre dado por el usuario hasta encontrar un salto de línea, lo cual sería poco eficiente, y más sabiendo que hay clases que hacen eso mismo.

Con que leer texto del teclado es algo común, seguro que hay alguna librería que tiene una clase que permite leer `Strings`. Buscando un poco encontraremos que lo más común es utilizar `java.io.BufferedReader`, y aquí nos aparece otro problema: ¿cómo creamos un objeto de esta clase?

En este momento se nos puede plantear la duda de si es necesario, ya que hasta ahora no hemos tenido que hacerlo, pero hay que tener en cuenta que nosotros estamos utilizando una variable de una clase (en este caso `System.out`), y esta variable ya está inicializada por el sistema. Podemos preguntarnos entonces por qué no tenemos que crear un objeto de la clase `System`. El motivo es que **la variable `System.out` es estática: eso significa que la comparten todos los objetos de la clase, y no hace falta instanciar ningún objeto específico para acceder a ella**, tal y como vimos con más detalle en el tema de la orientación a objetos.

Volvamos al problema original: ¿cómo obtenemos un objeto de la clase `BufferedReader`?

La solución está en utilizar una **función constructora: son funciones que generan un objeto de una clase determinada**. En general, podrían llamarse de cualquier forma, pero la sintaxis de Java requiere que se llamen igual que la clase. De todas formas, esto no presenta un inconveniente para poder tener varias funciones constructoras, ya que Java permite que dos funciones se llamen igual si tienen parámetros diferentes.

En concreto, nos interesa crear un `BufferedReader` a partir de la entrada estándar: no habría nada más fácil que pasarle `System.in` de parámetro a la función constructora, de no ser porque no existe ninguna constructora de esta clase que espere por parámetro un `InputStream`. En realidad, la única manera de crear un objeto de esta clase es a partir de un objeto de la clase `Reader`, y para sorpresa nuestra, mirando la documentación:

<https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html>

Veremos que nos encontramos con un nuevo problema aparentemente grave para nuestros propósitos: pese a que la clase `Reader` tiene constructores, no podemos utilizarlos ya que son de tipo `protected`, para mayor claridad, no son `public`. Además, para más inri, la clase está definida como `abstract`, lo cual significa que no se pueden crear/instanciar objetos de la misma.

Parece que hayamos llegado a un punto muerto y aquí es donde entra el concepto de herencia: cuando decimos que la función constructora de `BufferedReader` espera un parámetro de tipo `Reader`, nos referimos en realidad a cualquier objeto de esta clase, más los de cualquier clase que haya heredado a la clase `Reader`. En este caso, podemos ver que la clase `InputStreamReader` nos sirve completamente:

<https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html>

Por un lado, es una subclase de `Reader` (lo que equivale a decir que la hereda), y por el otro acepta como parámetro a su constructora un objeto de la clase `InputStream` que es justamente lo que tenemos: `System.in`.

Podemos pensar que esto es demasiado complicado sólo para leer de teclado, pero pensemos en qué hacemos cuando hacemos lo mismo en C. Probablemente utilizaríamos la función `fgets` pasándole

`stdin` como entrada. Internamente, esta función llama a la función `read`, guarda en un *buffer* interno una determinada cantidad de caracteres, y nos devuelve una cadena de este *buffer*.

En Java, hacemos lo mismo: sobre la entrada estándar (`InputStream`), ponemos un *buffer* (`BufferedReader`) y le pedimos una línea. El hecho de que tengamos que utilizar un objeto de la clase '`InputStreamReader`' es sólo para hacer de puente entre una y otra. Analicemos el programa como hemos hecho antes, pero para simplificar nos saltaremos lo ya hemos visto en el ejemplo anterior:

```
import java.lang.*;  
import java.io.*;  
public class ComoMeLlamo  
{  
    public static void main(String args[ ]) throws IOException  
    {  
        ...  
    }  
}
```

Hemos añadido un nuevo elemento en la función `main`, la cláusula `throws`, en la definición de la función, esto indica al compilador que dentro de esta podría producirse algún error o excepción que la función no captura (control de errores). Como nosotros no hacemos control de errores, en caso de error le pasamos la responsabilidad a la "entidad" superior lanzando el posible error a la función que nos llama para que ella lo trate. En este caso, la función `main` es llamada por la máquina virtual (JVM) cuando intenta ejecutar el programa, así que nadie capturaría la excepción y aparecería un error por pantalla.

Si probamos de quitar la cláusula `throws` y compilar el programa, averiguaremos qué línea de la función podría llegar a lanzar un error de entrada/salida (`IOException`). Java no permite que nos olvidemos de los errores por las buenas o por las malas. Debemos siempre tratarlos, ya sea directamente mediante su captura con `catch` (veremos más adelante cómo hacerlo), o los enviamos/lanzamos a la función que nos ha llamado.

```
import java.lang.*;  
import java.io.*;  
public class ComoMeLlamo  
{  
    public static void main(String args[ ]) throws IOException  
    {  
        InputStreamReader reader;  
        BufferedReader entrada;  
        String nombre;  
        ...  
    }  
}
```

Estas tres líneas son las definiciones de las variables locales de la función `main`. Ya hemos visto al inicio por qué utilizamos cada una de ellas. Si no, no pasa nada, podemos volver a revisarlo hasta estar seguros.

También hay que fijarse en el hecho de que Java no trata las cadenas de caracteres como un vector de caracteres acabado en '0', como hace C por ejemplo. Existe la clase `String`, que tiene un amplio abanico de funciones que podemos utilizar cuando las necesitemos y ayudan a trabajar de forma fácil con ellos.

ComoMeLlamo

```
import java.lang.*;
import java.io.*;
public class ComoMeLlamo
{
    public static void main(String args[ ]) throws IOException
    {
        InputStreamReader reader;
        BufferedReader entrada;
        String nombre;
        reader=new InputStreamReader(System.in);
        entrada=new BufferedReader(reader);
        ...
    }
}
```

Aquí entramos de lleno en cómo invocar constructores en Java: siempre utilizaremos la cláusula `new` delante de la llamada a la función constructora, y obtendremos un objeto de esa clase. Recordemos que, en Java, las funciones constructoras tienen que tener el mismo nombre que la clase.

ComoMeLlamo

```
import java.lang.*;
import java.io.*;
public class ComoMeLlamo
{
    public static void main(String args[ ]) throws IOException
    {
        InputStreamReader reader;
        BufferedReader entrada;
        String nombre;
        reader=new InputStreamReader(System.in);
        entrada=new BufferedReader(reader);
        System.out.print("¿Cómo te llamas? ");
        nombre=entrada.readLine();
        System.out.println("Te llamas "+nombre);
    }
}
```

La clase `BufferedReader` tienen una función `readLine` que devuelve un `String`. Esto nos permite leer lo que el usuario introduce por teclado como nombre de forma fácil:

<https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html>

Para imprimir por pantalla el nombre introducido utilizamos el hecho que el compilador de Java permite concatenar dos (o más) `Strings` con el operador suma ('+'). De hecho, el compilador traduce esta línea a algo bastante más complicado, pero no hace falta entrar en tantos detalles.

4.3. Punteros y asignación de memoria

Hemos visto dos ejemplos, y si nos fijamos, no hemos reservado ni liberado memoria en ningún momento como se haría en C. Java trabaja como lo hace por ejemplo Python, es el mismo Java quien administra de forma transparente el programador de memoria, se encarga de realizar la reserva de memoria para el objeto, automáticamente, cuando llamamos a la función constructora y cuando no se necesita el objeto y la memoria asociada a él, también se encarga de liberarla sin la intervención del programador.

Para la liberación/administración de la memoria, Java utiliza un proceso especializado que recibe el nombre de *Garbage Collection* (recolección de basura), cuyo significado es bastante autoexplicativo. Automáticamente, Java se encarga de ir liberando de la memoria los objetos que ya no se referencian. Cómo lo hace, es un tema que no vamos a tratar, ya que hay múltiples formas de hacerlo, y queda fuera de lo que se explica aquí. Pero de manera aproximada aplica una algoritmo de optimización para predecir qué elementos en la memoria ya son antiguos y se pueden descartar y qué otros elementos es probable que se vuelvan a utilizar y vale la pena conservar, con eso realiza una ordenación por prioridad, en función de esta los elimina cuando necesita más memoria o cuando está bastante seguro que no serán necesarios durante el resto de la ejecución del programa.



Figura 16. Garbage Collection, auto reciclaje. Recuperado de: <https://es.pngtree.com/free-png-vectors/contenedores-de-reciclaje>

Respecto a los punteros, no existen en Java, hecho que facilita la programación. Pero hay que tener en cuenta, de todas formas, que todos los objetos se pasan por referencia a las funciones. Siendo equivalente al paso de parámetros por punteros, como se hace por ejemplo en C. Esto significa que, si en el ejemplo anterior la constructora de BufferedReader modificara el Reader que le pasamos por parámetro, por ejemplo y llamaría a su función close, afectaría a todo el programa, no sólo a la función. Para la clase podría compararse con una estructura en C, para las funciones sería como decir que estas reciben los parámetros como un puntero a esa estructura, no una copia de la misma. De todas formas, sólo hay que tenerlo presente, no preocuparse por ello, ya que, de hecho, en Java no vamos a ver punteros por ninguna parte, ya que no existen explícitamente.

Tenemos, como en la mayoría de lenguajes, unos tipos primitivos que en el caso del Java tienen la particularidad que no son objetos. En la siguiente tabla se describen los tipos primitivos presentes en Java:

Tabla 1
Tipos primitivos de Java

Nombre	Declaración	Memoria requerida	Rango
Booleano	boolean	1 bit	true - false
Byte	byte	1 byte (8 Bits)	[-128 .. 127]
Entero pequeño	short	2 byte (16 Bits)	[-32,768 .. 32,767]
Entero	int	4 byte (32 Bits)	[-2 ³¹ .. 2 ³¹ -1]
Entero largo	long	8 byte (64 Bits)	[-2 ⁶³ .. 2 ⁶³ -1]
Real	float	4 byte (32 Bits)	[±3,4·10 ⁻³⁸ .. ±3,4·10 ³⁸]
Real largo	double	8 byte (64 Bits)	[±1,7·10 ⁻³⁰⁸ .. ±1,7·10 ³⁰⁸]
Carácter	char	2 byte (16 Bits)	['\u0000' .. '\uffff'] o [0 .. 65.535]

Nota. Elaboración propia.

Estos tipos primitivos se pasan siempre por valor. Serán variables locales a las funciones que los utilicen o, dicho de otra manera, las funciones a las que se los pasamos por parámetro trabajarán con una copia de los mismos. No es posible pasar referencias a estos objetos, ya que, como hemos dicho anteriormente, no hay forma de forzar en Java que algo sea un puntero o trabajar con punteros.

4.4. Sintaxis básica de Java

La sintaxis de Java es muy parecida al C, de todas formas, vamos a repasarla para aquellos que no la conozcan. Como en el caso el Python no tenemos punteros (apuntadores) y el Java es un poco más sensible para realizar las conversiones automáticas entre tipos que otros lenguajes, es recomendable indicárselas de forma explícita para evitar problemas indicando entre paréntesis la clase que queremos que se realice la conversión. Por ejemplo, para convertir un double a float:

```
double n1;
float n2;
n1=5.6;
n2=(float) n1;
```

Sin el `(float)`, C quizá nos daría un *warning* (alerta de posible error) al intentar compilar; Java, simplemente, no nos lo permitirá y la compilación fallará. Se asegura de que realmente estamos haciendo lo que queremos hacer.

Otro punto a tener en cuenta, también relacionado con éste, es que no podemos utilizar un entero para representar un valor booleano. El típico:

```
int i;  
if(i)  
...
```

En Java no podemos hacerlo, tenemos que hacer la comparación que será la que nos regrese el valor booleano:

```
int i;  
if(i != 0)  
...
```

Teniendo en cuenta estos detalles, podemos hacer tranquilamente nuestro programa usando lo que ya conocemos: `while`, `for`, `switch/case`, `if`,...

Veamos más detalles sobre cómo funciona el Java. Veamos mediante otro ejemplo como trabajar con los parámetros de entrada:

MisParametros

```
public class MisParametros  
{  
    public static void main(String args[ ])  
    {  
        int i;  
        if(args.length == 0)  
        {  
            System.out.println("Necesito al menos un parámetro.");  
            return;  
        }  
        for(i=0 ; i < args.length ; i++)  
        {  
            System.out.println("Parámetro "+i+": "+args[i]);  
        }  
    }  
}
```

En este programa podemos ver que los vectores tienen un atributo `length`, que, como su nombre indica, nos indica su longitud.

Además, el objeto nulo es `null` en minúsculas, al contrario de por ejemplo C, que es `NULL` (puntero nulo).

Recordemos que en Java todas las variables que guardemos serán punteros sin que lo sepamos, excepto los tipos primitivos (`int`, `double`, `boolean`, ...).

Para acabar, veamos cómo crear un vector/array en Java. Definiremos un vector de enteros:

```
int vectordeenteros[];  
vectordeenteros=new int[25];
```

Como podemos ver, definimos `vectordeenteros` como vector (utilizando `[]`), pero no decimos su tamaño en la declaración. Más adelante, podemos reservar espacio para este vector indicando su longitud (en este caso, 25 elementos).

Tema 5. Depuración de programas

Los diferentes entornos de desarrollo gráfico con los que podemos trabajar incorporan casi siempre un depurador con interfaz gráfica, con funcionalidades parecidas o superiores a las vistas, por ejemplo, en los cursos iniciales de programación. Solo en caso de necesidad necesitaremos utilizar el depurador usando la consola. Es una herramienta básica que siempre tendremos a mano para sacarnos de un apuro, cuando los arcaicos chivatos de imprimir valores de las variables o los puntos por donde pasa la ejecución por la salida estándar no sean suficientes para depurar los errores de nuestro programa.

El *debugger* permitirá ejecutar nuestro programa paso a paso, línea por línea de código, permitiendo que en cada una de ellas podamos consultar los valores actuales de las variables e ir viendo si el programa realiza lo que queremos. Conque hacerlo todo paso a paso es poco eficiente en programas extensos, uno de los elementos básicos es poder indicar en qué puntos del código queremos parar, para realizar una verificación, es poner puntos de parada (*BreakPoints*) y de esta forma poder ejecutar un bloque entero de código de forma continua hasta llegar a ese punto. Es importante señalar que los puntos de parada sólo se pueden poner en líneas de código donde hagamos una acción/operación/cálculo, por ejemplo, una asignación a una variable.

También la mayoría de ellos incluyen la posibilidad de saltarnos o entrar en las diferentes funciones, para de igual forma que los puntos de parada poder ir de forma más rápida al punto conflictivo.

La mayoría de depuradores integrados en los entornos de desarrollo integrados, IDEs por sus siglas en inglés (*Integrated Development Environment*), tienen las funcionalidades indicadas. En el siguiente apartado se repasarán, a modo de ejemplo, las que ofrecen el depurador integrado en NetBeans.

5.1 Depurador en NetBeans

Podemos utilizar la interfaz gráfica o por comando, pero las funcionalidades son las mismas. En la Figura 17 podemos ver la captura de los diferentes botones disponibles:

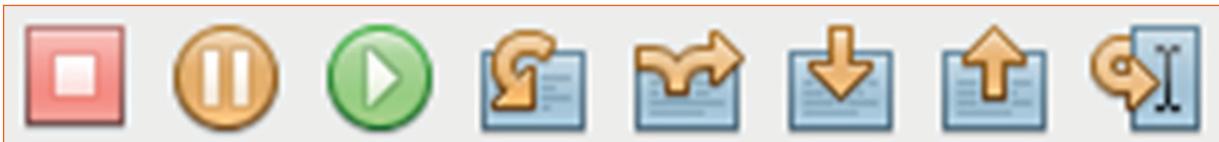


Figura 17. Interface gráfica depurador NetBeans. Fuente: elaboración propia.

Nombre de la opción / (atajo de teclado a la misma) / (descripción del ícono)

- *Finish Debugger Session* / (Shift+F5) / (Icono cuadrado rojo): terminar la depuración del programa.
- *Pause* / (Icono pausa): pausar ejecución.
- *Continue* / (F5) / (Icono verde flecha): reanuda la ejecución hasta conseguir un punto de parada (*BreakPoint*) o terminar el programa.
- *Step Over* / (F8): ejecuta una sola línea de código. Si la instrucción es una llamada a una función, ejecuta la función completa como un todo, sin entrar en sus diferentes líneas de código.
- *Step Over Expression* / (Shift+F8): ejecuta una sola línea de código. Si la instrucción es una expresión compleja, por ejemplo con llamadas a varias funciones, permite ir paso a paso por los diferentes elementos de la expresión, pudiendo consultar su parámetros de llamada y sus retornos.
- *Step Into* / (F7): ejecuta una sola línea de código. Si la instrucción es una llamada a una función, entra dentro de la función y procede ejecutando la primera línea dentro de la función.
- *Step Out* / (Ctrl+F7): ejecuta una sola línea de código. Si la instrucción está dentro de una función, ejecuta el resto de líneas de la función y regresa al punto donde fue llamada.
- *Run to Cursor* / (F4): se ejecuta el programa hasta la instrucción donde se encuentra el cursor.

Si trabajamos con la IDE podemos ir observando sobre el código la línea de código en la que actualmente nos encontramos resaltada en verde, igualmente veremos resaltadas en rojo las líneas de código donde hemos puesto un punto de parada (*BreakPoint*). Según vayamos avanzando por el código el editor irá saltando entre los diferentes ficheros de código de las diferentes clases de ser necesario. Podemos ver las partes resaltadas en verde y rojo en la Figura 18:

```

20  /*
21   * 
22  public static void main(String[] args)
23  {
24      // TODO code application logic here
25      Scanner sn = new Scanner(System.in);
26      boolean salir = false;
27      int opcion;
28      double n1, n2;
29      //Suma suma;
30      //Resta resta;
31      Operacion operacion;
32
33      while (!salir) {
34          System.out.println("1) Suma");
35          System.out.println("2) Resta");
36          System.out.println("3) Multiplicar");
37          System.out.println("4) Elecciones");
38          System.out.println("5) Resultados");
39          System.out.println("6) Centros");
40          System.out.println("7) Salir");
41
42          System.out.print("Elija una opción: ");
43          String op = sn.nextLine();
44
45          if (op.equals("1"))
46              suma();
47          else if (op.equals("2"))
48              resta();
49          else if (op.equals("3"))
50              multiplicar();
51          else if (op.equals("4"))
52              elecciones();
53          else if (op.equals("5"))
54              resultados();
55          else if (op.equals("6"))
56              centros();
57          else if (op.equals("7"))
58              salir = true;
59      }
60  }
61
62  */

```

Figura 18. Línea actual de código (verde) y futuro punto de parada (rojo). Fuente: elaboración propia.

Para poder visualizar los datos (tipo, valor, etc.) en tiempo real de las diferentes variables tenemos una ventana dentro de la IDE *variables* donde encontramos las variables actuales en memoria del programa. En la Figura 19 podemos observar un ejemplos, donde observamos que la variable *salir* que es de tipo boolean en estos momentos tienen el valor *false*.

	Name	Type	Value
...	<Enter new watch>		...
...	Static		...
...	args	String[]	#87(length=0)
...	sn	Scanner	#249
...	salir	boolean	false

Figura 19. Ventana con la información de las variables actuales en memoria. Fuente: elaboración propia.

Tema 6. Documentación

Documentar el código que realizamos es necesario sea cual sea el tamaño del sistema que estamos implementando. Una buena documentación nos evitará dolores de cabeza en el futuro como hemos visto desde el principio del grado en diversas asignaturas. Oracle desarrollo la utilidad de Javadoc para la generación automatizada de la misma, la cual se utiliza normalmente como el estándar en la mayoría de industrias para documentar las clases de Java. Generando un documentación básica de las clases y sus funciones (APIs) usando el formato HTML, permitiendo su consulta rápida en múltiples plataformas. Como en el caso del depurador, la mayoría de entornos de desarrollo integran dicha utilidad para hacer su uso y consulta más fácil.

6.1 Javadoc

Javadoc es una utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java. Javadoc es el estándar de la industria para documentar clases de Java. La mayoría de los IDEs los generan automáticamente, por ejemplo, en NetBeans usando la opción de *Generate Javadoc*.

Para que las IDEs puedan generar la documentación debemos usar diferentes claves dentro de nuestro código para indicar los diferentes elementos que queramos documentar, en la Tabla 2 podemos ver las diferentes opciones:

Tabla 2

Diferentes claves para indicar lo que queremos documentar a Javadoc.

Clave	Descripción
@author	Nombre/s de los autores.
@param	Definición de los parámetros del método, requerido para todos ellos.
@return	Definición de que regresa el método.
@see	Asociación con otro método o clase. Referencia: (#método(); clase#método(); paquete.clase; paquete.clase#método())
@version	Versión
@deprecated	Marca un elemento obsoleto, mejor no usar

Nota. Elaboración propia.

Las claves `@authory` `@version` son solo para documentar clases e interfaces. No son válidas en cabecera de constructores ni métodos.

La clave `@param` es solo para documentar constructores y métodos.

La clave `@return` es solo para documentar métodos de tipo función.

Dentro de los comentarios se admite HTML. Por ejemplo, se puede referenciar una página web como un enlace.

Ejemplo:

```
package calculadorasr;
/**
 *
 * Esta clase implementa la operación multiplicar
 * @author: Roger
 * @version: v1.0 25/11/2019
 * @see <a href="https://www.universidadviu.com/"> VIU </a>
 */
public class Multiplicar extends Operacion {
/**
 * Constructor de la clase
 * @param num1 Primer número a multiplicar
 * @param num2 Segundo número a multiplicar
 */
public Multiplicar(double num1, double num2)
{
    super(num1, num2);
}
/**
 * Regresa la multiplicación de los dos números definidos en el constructor
 * @return Resultado de la multiplicación
```

```
 */
protected double operar()
{
    return n1*n2;
}
/**
 * Regresa el operador matemático de la operación como texto
 * @return En el caso de la multiplicación '*'
 */
public char operador()
{
    return '*';
}
```


Tema 7. Conceptos básicos de diseño de interfaces

Hasta ahora todos los ejemplos han manejado la interacción con el usuario usando la consola de texto, pero una forma mucho más agradable y visual de interacción es hacerlo mediante una interface gráfica de usuario. El hecho de agregar a nuestros programas un entorno gráfico no modifica la propiedad que tiene Java de ser independiente de la plataforma donde se ejecuta.

Hay diversos paquetes para dotar a nuestros programas de una interfaz gráfica. En este manual se incluye una breve reseña de Java Abstract Window Toolkit, pero eso no significa que otras opciones no sean totalmente validas, como por ejemplo Swing:



Tutorial de Swing

<http://dis.um.es/~bmoros/Tutorial/introduccion/indice2.html#veintiuno>



Documentación Package Swing

<https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html>

7.1. Java Abstract Window Toolkit (Java AWT)

Conjunto de herramientas para crear una interfaz gráfica de usuario. Incluye todos los elementos necesarios y ha estado disponible desde 1995. Hace que nuestras aplicaciones parezcan "nativas" a la plataforma donde se están ejecutando, por ejemplo, las ventanas se verán como si fueran ventanas

Windows o Linux según el sistema operativo. Esta adaptación a la plataforma puede no gustar, hay desarrolladores que prefieren que su aplicación se vea exactamente igual independientemente del sistema operativo donde se ejecute.



Documentación Package AWT

<https://docs.oracle.com/javase/8/docs/api/java/awt/package-summary.html>

Podemos encontrar varios elementos básicos que combinados nos permitirán disponer de una completa interfaz gráfica. Por ejemplo: Component, Container, Dialog, Button, Event, Canvas, etc. Fíjémonos en las piezas básicas:

- Clase Component: se trata de la clase que hereda todo aquello que pueda ser mostrado en una ventana.
- Clase Container: subclase de Component, que heredará todas las clases que vayan a hacer de "contenedores", díganse ventanas, marcos, paneles, etc.

Entre los componentes encontraremos botones, campos de texto y casi todo lo que necesitaremos en la mayoría de aplicaciones.

Veamos un ejemplo del HolaMundo, pero esta vez usando ventanas:

HolaMundo

```
import java.awt.*;
public class HolaMundo
{
    public static void main(String args[ ])
    {
        Frame ventana=new Frame("Ventana - Hola Mundo");
        Label hola=new Label("¡Hola, Mundo!", Label.CENTER);
        ventana.setSize(150, 100);
        ventana.add(hola);
        ventana.setVisible(true);
    }
}
```

Este código mostrará una pequeña ventana con el texto ¡Hola, Mundo!, como podemos ver en la Figura 20:

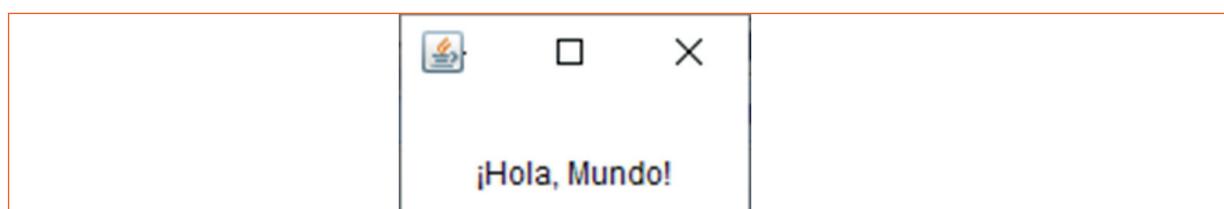


Figura 20. Venta Hola Mundo gráfica. Fuente: elaboración propia.

Regresemos al código y veamos elemento por elemento:

```
Frame ventana=new Frame("Ventana - Hola Mundo");
```

Permite crear un nuevo marco/ventana (frame)

```
Label hola=new Label("¡Hola, Mundo!", Label.CENTER);
```

Creamos una nueva etiqueta/texto. Vemos que uno de los parámetros hace una referencia a un valor definido internamente, para evitar que tengamos que memorizar todas las posibles posiciones que están codificadas como enteros, podemos usar esta definición interna. En este ejemplo Label.CENTER equivale al int 0, que centrará el texto de la etiqueta, mucho más intuitivo que recordar que centrar equivale a 0.

```
ventana.setSize(150, 100);
```

Definimos el tamaño en pixeles de la ventana.

```
ventana.add(hola);
```

Añadimos el texto a la ventana.

```
ventana.setVisible(true);
```

Por último, hacemos visible la ventana.

Vamos a complicar un poco el HolaMundo, ¿cómo hacemos si queremos añadir por ejemplo botones?
Veamos este otro ejemplo:

Botones

```
import java.awt.*;  
public class Botones extends Panel  
{  
    public Botones()  
    {  
        LayoutManager layout=new FlowLayout();  
        Button boton1=new Button("1");  
        Button boton2=new Button("2");  
        Button boton3=new Button("3");  
        TextField text=new TextField("Esto es una botonera");  
        text.setEditable(false);  
        this.setLayout(layout);  
        this.add(boton1);  
        this.add(boton2);  
        this.add(boton3);  
    }  
}
```

```

        this.add(text);
    }
    public static void main(String args[ ])
    {
        Frame ventana=new Frame("Ventana - Menu Botones");
        Botones botones=new Botones();
        ventana.setSize(280, 70);
        ventana.add(botones);
        ventana.setVisible(true);
    }
}

```

Veamos los elementos más relevantes:

```
public class Botones extends Panel
```

Panel es un tipo de contenedor (Container) básico.

```
LayoutManager layout=new FlowLayout();
```

El LayoutManager es un gestor para posicionar los componentes dentro del Container de forma automática. Si no, toca posicionar los elementos 1 a 1 de forma manual, teniendo más control pero siendo más farragoso de programar.

```
Button boton1=new Button("1");
```

Crea un botón con el texto "1".

El resto de elementos son equivalentes al primer ejemplo. En la Figura 21 podemos observar el resultado de ejecutar este código de ejemplo:

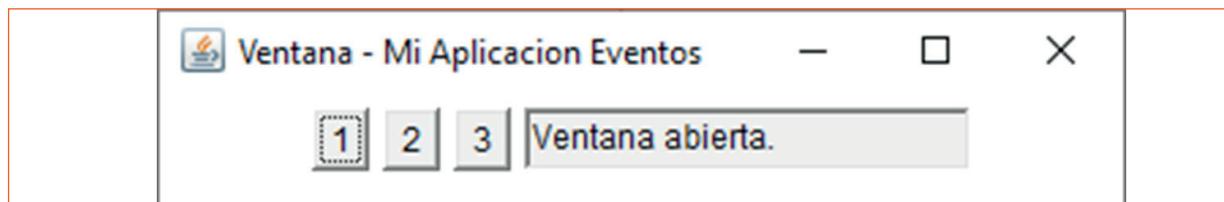


Figura 21. Venta con botones y cuadro de texto. Fuente: elaboración propia.

Con estos dos ejemplos hemos visto la parte gráfica, pero ¿Cómo interactuamos? Es decir, ¿cómo detectamos las acciones del usuario?

Para poder detectar qué hace el usuario usaremos el paradigma de la programación orientada a eventos:

Evento = acción por parte del usuario sobre la interfaz gráfica.

Por ejemplo, si nos interesa tratar en un objeto los eventos de pulsación de un botón, añadiremos la clase a la lista de receptores del mismo definiéndola como receptora (*Listener*). Cuando ocurra la pulsación del botón, se nos notificará mediante la ocurrencia de un evento (*ActionEvent*). En ese momento debemos podemos tratar el evento según nuestras necesidades del programa. Como en los casos anteriores veámoslo con un ejemplo:

Botones2

```
import java.awt.*;
import java.awt.event.*;
public class Botones2 extends Panel implements ActionListener
{
    private TextField text;
    public Botones2()
    {
        LayoutManager layout=new FlowLayout();
        Button boton1=new Button("1");
        Button boton2=new Button("2");
        Button boton3=new Button("3");
        text=new TextField("Esto es una botonera");
        text.setEditable(false);
        this.setLayout(layout);
        this.add(boton1);
        this.add(boton2);
        this.add(boton3);
        this.add(text);
        boton1.addActionListener(this);
        boton2.addActionListener(this);
        boton3.addActionListener(this);
    }
    public void mensaje(String mensaje)
    {
        text.setText(mensaje);
        System.out.println(mensaje);
    }
    public void actionPerformed(ActionEvent event)
    {
        String accion=event.getActionCommand();
        text.setText(accion);
    }
}
```

MiAplicacionEventos

```

import java.awt.*;
import java.awt.event.*;
public class MiAplicacionEventos extends Frame implements WindowListener {
    Botones2 botones;
    public MiAplicacionEventos()
    {
        super("Ventana - Mi Aplicacion Eventos");
        botones=new Botones2();
        this.setSize(280, 65);
        this.add(botones);
        this.addWindowListener(this);
    }
    public void windowActivated(WindowEvent event)
    {
        botones.mensaje("Ventana activada.");
    }
    public void windowDeactivated(WindowEvent event)
    {
        botones.mensaje("Ventana desactivada.");
    }
    public void windowIconified(WindowEvent event)
    {
        botones.mensaje("Ventana minimizada.");
    }
    public void windowDeiconified(WindowEvent event)
    {
        botones.mensaje("Ventana desminimizada.");
    }
    public void windowOpened(WindowEvent event)
    {
        botones.mensaje("Ventana abierta.");
    }
    public void windowClosing(WindowEvent event)
    {
        botones.mensaje("Ventana cerrandose.");
        this.dispose();
    }
    public void windowClosed(WindowEvent event)
    {
        botones.mensaje("Ventana cerrada.");
        System.exit(0);
    }
}

```

```
}

public static void main(String args[])
{
    MiAplicacion app=new MiAplicacion();
    app.setVisible(true);
}
}
```

Veamos, como en los otros ejemplos, las líneas de código más relevantes por separado:

```
public class Botones2 extends Panel implements ActionListener
```

Herencia múltiple usando `extends` e `implements`. Por un lado, tenemos que queremos un `Panel` y por el otro que este debe responder a eventos y por eso implementará la clase `ActionListener`, haciendo uso de la "herencia múltiple" en Java.

```
public void mensaje(String mensaje)
{
    text.setText(mensaje);
    System.out.println(mensaje);
}
```

Mostrar un texto por consola.

```
public void actionPerformed(ActionEvent event)
{
    String accion=event.getActionCommand();
    text.setText(accion);
}
```

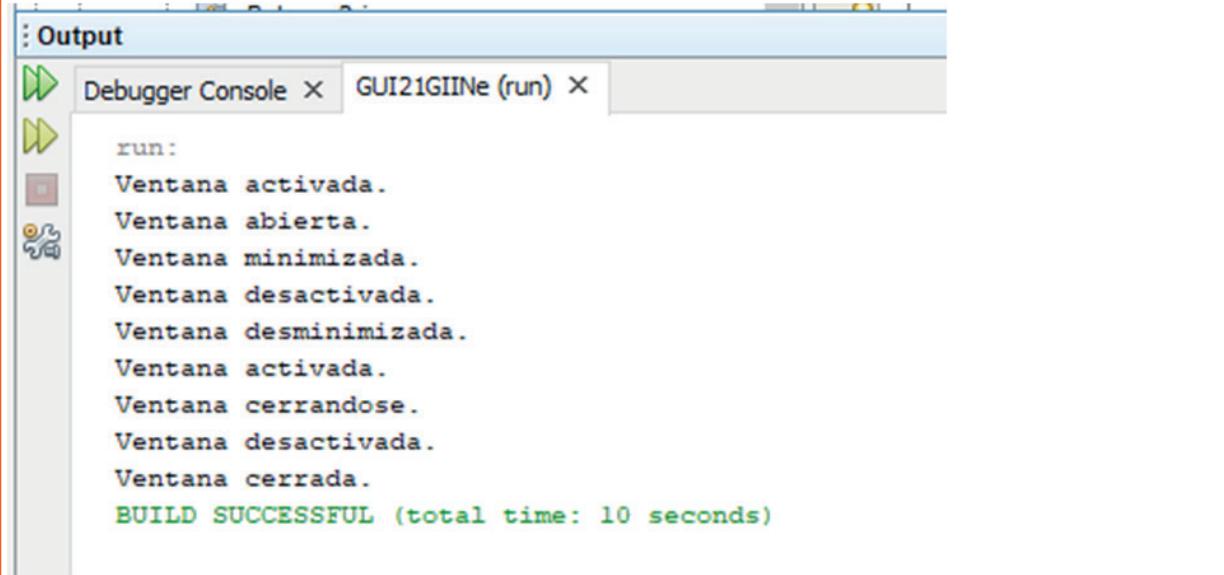
Mostrar un texto por el cuadro de texto de la ventana.

```
public MiAplicacionEventos()
...
    this.addWindowListener(this);
```

La clase `MiAplicacionEventos` debe tratar los eventos producidos en la ventana, por ese motivo es necesario que se "registre" como "escuchador" de eventos.

```
public void windowActivated(WindowEvent event)
{
    botones.mensaje("Ventana activada.");
}
```

Para cada posible evento, en este caso la activación de la ventana, debemos implementar que acción realizar. En la Figura 22 podemos observar los mensajes producidos por diferentes eventos.



The screenshot shows an IDE's Output window titled "Output". It contains two tabs: "Debugger Console" and "GUI21GIINe (run)". The "run" tab is active and displays the following log messages:

```
run:  
Ventana activada.  
Ventana abierta.  
Ventana minimizada.  
Ventana desactivada.  
Ventana desminimizada.  
Ventana activada.  
Ventana cerrandose.  
Ventana desactivada.  
Ventana cerrada.  
BUILD SUCCESSFUL (total time: 10 seconds)
```

Figura 22. Eventos mostrados por consola. Fuente: elaboración propia.

Ahora ya tenemos una ventana para mostrar gráficamente nuestro programa y que es capaz de reaccionar a las acciones (eventos) del usuario.

Tema 8. Ejemplo: calculadora básica con base de datos

Para integrar todo lo visto en este manual y dar un ejemplo de conectividad con una base de datos, a continuación, se detallará un ejemplo de una calculadora básica con interfaz gráfica usando AWT y que guardará un histórico de las operaciones en una base de datos alojada en **Amazon Web Services** (AWS).

8.1. Amazon Web Services Educate

En primer lugar y antes de empezar a programar será necesario darnos de alta en *Amazon Web Services* (AWS). Este servicio en la nube de Amazon permite la creación de una cuenta gratuita para estudiantes, entre los muchos servicios que ofrece hay el de tener una base de datos alojada en dicho servicio.

En primer lugar, será necesario acceder a la web:



AWS

<https://aws.amazon.com/es/education/awseducate/>

Donde encontraremos mucha información del servicio ofrecido y sus posibilidades. Una vez allí hay que darse de alta (registrarse), en el momento de la creación de este manual el link directo al registro era:



AWS/registrarse

https://www.awseducate.com/registration#APP_TYPE

En la Figura 23 podemos observar la pantalla de bienvenida al registro, en ella podemos cambiar el idioma y debemos seleccionar el rol (tipo de usuario). En nuestro caso pediremos una cuenta como estudiante.

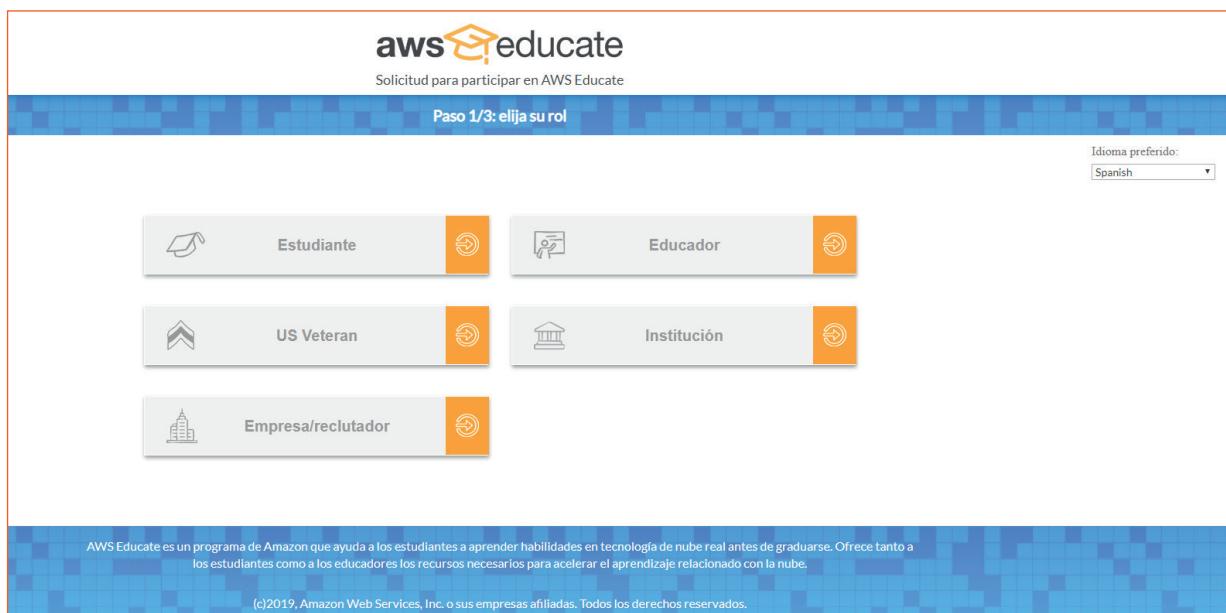


Figura 23. Paso 1 del registro en AWS. Fuente: elaboración propia.

En la Figura 24 podemos observar la pantalla donde nos piden nuestros datos. Es importante indicar la Universidad Internacional de Valencia y a ser posible usar nuestro email de estudiante, eso agilizará la autorización del registro para que podamos usar AWS. Nos pedirán un estimado de nuestra fecha de graduación, no hace falta que sea precisa, pero si debe ser posterior a la fecha actual.

Solicitud para participar en AWS Educate

Paso 2/3: háblenos de usted

Universidad Internacional de Valencia

Nombre

Apellido

Correo electrónico

Mes de graduación

Año de graduación

Mes de nacimiento

Año de nacimiento

Código promocional

Idioma preferido: Spanish

No soy un robot

reCAPTCHA

Please note that any personal information you provide will be treated in accordance with the AWS Educate Terms and Conditions and AWS Privacy Notice

[SIGUIENTE](#)

Figura 24. Paso 2 del registro en AWS. Fuente: elaboración propia.

Para finalizar, en la Figura 25 podemos observar la pantalla donde nos piden aceptar las condiciones del servicio.

aws educate

Apply to join AWS Educate

Terms & Conditions

Preferred Language: Spanish

AWS EDUCATE TERMS AND CONDITIONS
(Last Updated April 30, 2019)

1.0 YOUR AGREEMENT WITH AWS

1.1 This Agreement. This set of terms and conditions (this "Agreement") is an agreement between you (or the Entity you work for) ("you") and Amazon Web Services, Inc. or other entity noted in Section 10 (in either case, "AWS", "we" or "us"). This Agreement governs your participation in the AWS Educate Program (the "Program") described at <https://aws.amazon.com/education/awseducate/> and its subpages (the "Program Site"), including (a) your use and submission of data, text, audio, video, images, software (including machine images), or other materials (collectively, "Content") in connection with the Program; and (b) your use of any tools, websites, and services AWS may provide to you in connection with the Program (collectively, the "Educate Tools"). If you are entering into this Agreement for a commercial entity, government institution, or any other entity ("Entity"), such as the company or educational institution you work for, you represent that you have legal authority to bind that Entity, and references to "you" in this Agreement will be deemed as referring to that Entity. If you have an AWS Customer Agreement (available at <http://aws.amazon.com/agreement/>) or other agreement between you and AWS governing your use of AWS services ("AWS Services Agreement"), that agreement will govern your use of the web services described in the Service Terms of the agreement and any other Service Offerings covered by (and as defined) therein.

You must scroll through the entire Terms and Conditions before accepting or declining.

Figura 25. Paso 3 del registro en AWS. Fuente: elaboración propia.

Una vez aceptadas sólo resta esperar un tiempo prudencial (normalmente menos de 1 semana) para que Amazon nos autorice nuestra cuenta en *AWS Educate*. Si la aceptación se demora excesivamente pueden pedirle a su profesor que los agregue a una aula creada en AWS. Para poder hacer esto él debe haber pedido una cuenta de educador para la cual piden más datos y la verificación es más lenta, y después configurar un aula para el curso.

Una vez tengamos la autorización de la cuenta deberemos entrar para poder crear una servidor con una base de datos. Inicialmente Amazon nos otorgará un saldo en dólares para poder usar durante 1 año de forma gratuita sus servicios. Para que ese saldo inicial nos rinda (sea suficiente para la asignatura), debemos seleccionar los servidores más económicos y en la ubicación más económica.

En el momento de creación de este manual, la ubicación más económica era el *cluster* de servidores en Virginia del Norte en Estados Unidos, como para el ejemplo no es relevante la latencia no es necesario elegir un *cluster* de servidores más cercano a nuestra ubicación (normalmente los *clusters* en Europa o Asia son más costosos).

Para el servidor de base de datos usaremos el servicio RDS, con una base de datos MariaDB (una versión de la base de datos MySQL con licencia de software libre). Adicionalmente usaremos un servidor que esté en un *Free tier* el cual tendrá un coste de operación por hora más económico. El rendimiento no será muy elevado, pero para este ejemplo será más que suficiente.

Si en algún momento decidimos usar AWS para un aplicación en producción que requiera menor latencia y más recursos a nivel de servidor, tenemos muchas opciones para ir escalando, pero los costes también lo hacen y con el saldo inicial de la cuenta de educación sólo podríamos mantenerlo por poco tiempo en funcionamiento. Siempre nos queda la opción de pagar con la tarjeta de crédito si realmente necesitaremos esas prestaciones.

A continuación, veremos los diferentes pasos para realizar la configuración al detalle. Estos pueden cambiar un poco, pero las configuraciones básicas a realizar serán más o menos las mismas. En caso de que los pasos sean muy diferentes preguntén a su profesor.

En primer lugar, una vez autentificados entraremos en la consola de administración de AWS (ver Figura 26). En ella deberemos buscar el apartado *Base de Datos* y seleccionar RDS, ver Figura 27.

The screenshot shows the AWS Management Console homepage. At the top, there's a navigation bar with the AWS logo, 'Servicios', 'Grupos de recursos', and a search bar. Below the search bar, there are several service links: 'Todos los servicios' (with EC2, Lightsail, ECR, ECS, EKS, Lambda, Batch, Elastic Beanstalk, Serverless Application Repository), 'Satélite Ground Station', 'Quantum Technologies Amazon Braket', 'Administración y gobierno AWS Organizations', 'Seguridad, identidad y conformidad IAM Resource Access Manager Cognito Secrets Manager GuardDuty Inspector Amazon Macie AWS Single Sign-On', and 'Acceda a los recursos desde cualquier lugar' (with a mobile phone icon and text about the AWS mobile app). On the right, there are sections for 'Explorar AWS' (Amazon Redshift) and 'Ejecute contenedores sin servidor con AWS Fargate'.

Figura 26. Paso 1 creación Base de Datos en AWS. Fuente: elaboración propia.

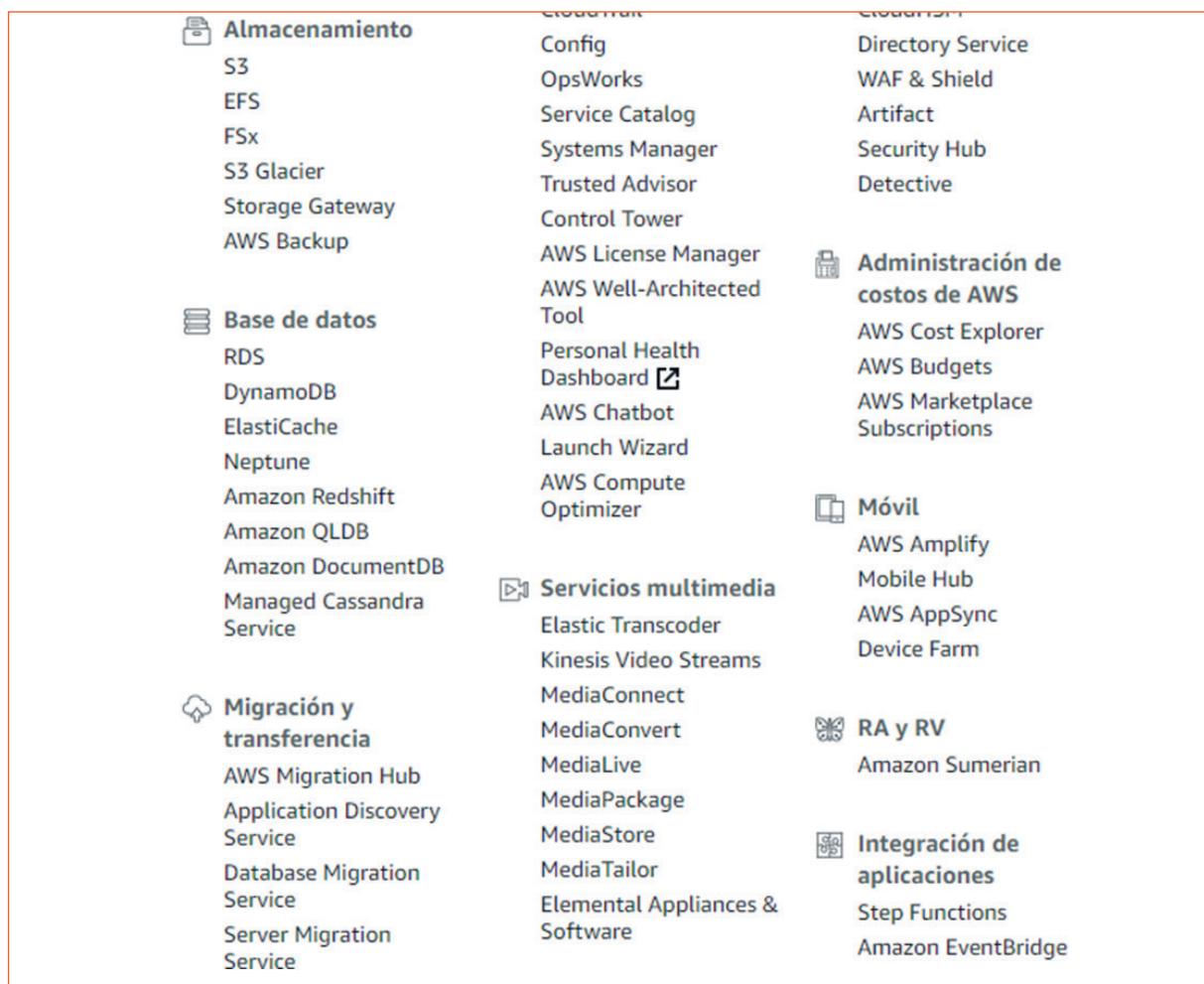


Figura 27. Paso 2 creación Base de Datos en AWS. Fuente: elaboración propia.

Una vez seleccionado RDS, deberemos buscar la opción de crear un base de datos (*Create database*), tal y como se muestra en la Figura 28.

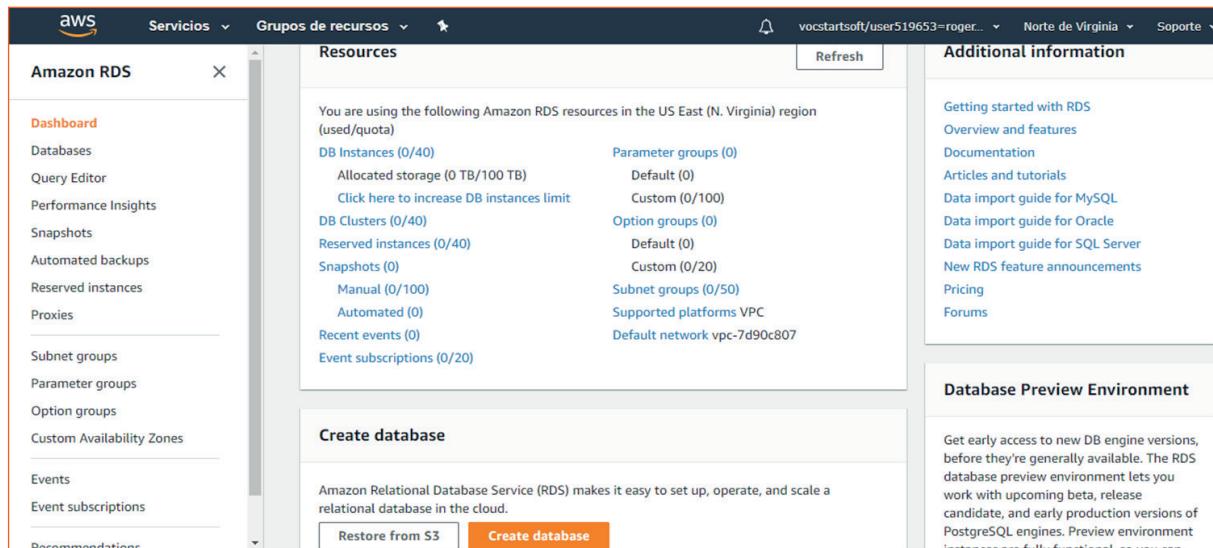


Figura 28. Paso 3 creación Base de Datos en AWS. Fuente: elaboración propia.

Como no requerimos ningún tipo de configuración especial, podemos seleccionar creación fácil (*Easy Create*) y como motor de la base de datos MariaDB (*Engine type: MariaDB*). Con las opciones por defecto será más que suficiente en nuestro caso. Podemos ver ambas opciones seleccionadas en la Figura 29.

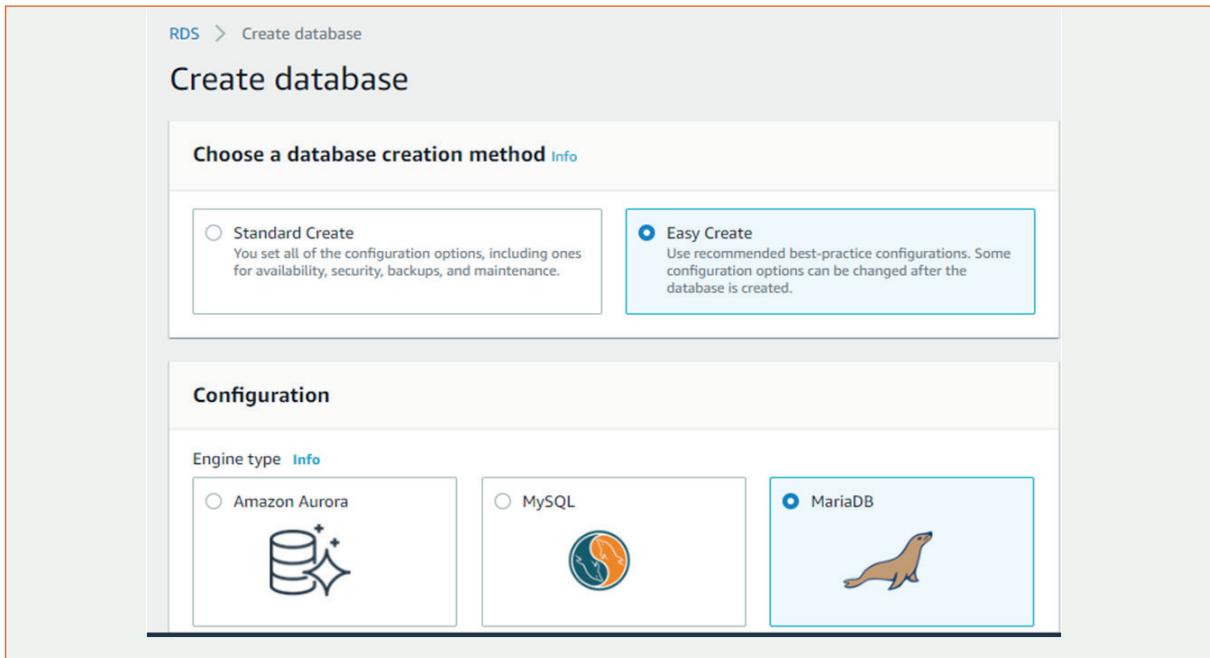


Figura 29. Paso 4 creación Base de Datos en AWS. Fuente: elaboración propia.

A continuación, deberemos seleccionar el tamaño de la instancia (*DB instance size*), que equivale a la cantidad de recursos del servidor. Como hemos comentado anteriormente, para hacer rendir el saldo en dólares, es recomendable utilizar un equipo *Free tier*. Adicionalmente nos pedirá los datos básicos como son el nombre y el usuario administrador. En la Figura 30 podemos ver los detalles.

DB instance size		
<input type="radio"/> Production db.r4.xlarge 4 vCPUs 30.5 GiB RAM 500 GiB 2.078 USD/hour	<input type="radio"/> Dev/Test db.r4.large 2 vCPUs 15.25 GiB RAM 100 GiB 0.256 USD/hour	<input checked="" type="radio"/> Free tier db.t2.micro 1 vCPUs 1 GiB RAM 20 GiB 0.020 USD/hour

DB instance identifier
 Type a name for your DB instance. The name must be unique cross all DB instances owned by your AWS account in the current AWS Region.
OCT19-21GIN

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens (1 to 15 for SQL Server). First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

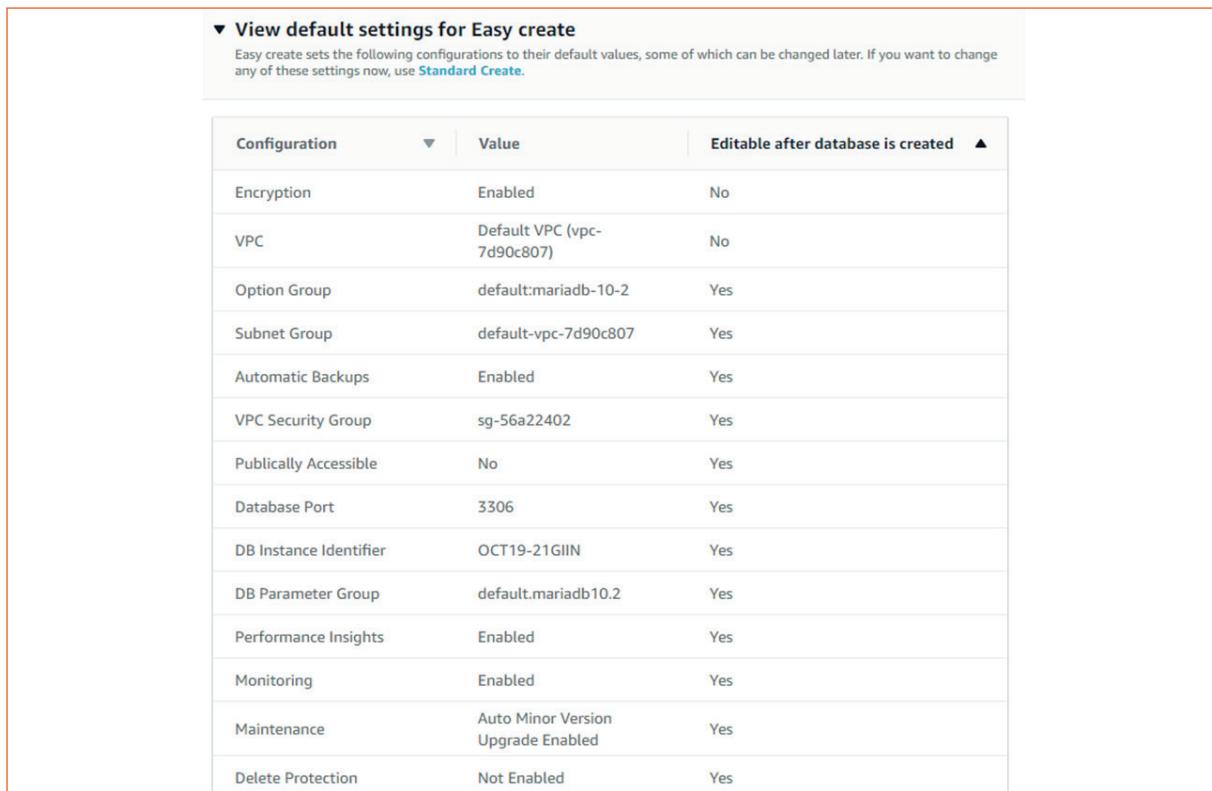
Master username [Info](#)
 Type a login ID for the master user of your DB instance.
admin

1 to 16 alphanumeric characters. First character must be a letter

Auto generate a password
 Amazon RDS can generate a password for you, or you can specify your own password

Figura 30. Paso 5 creación Base de Datos en AWS. Fuente: elaboración propia.

Es importante fijarse en el puerto por defecto, 3306. Si tuviéramos alguna limitación por un cortafuegos en el lugar donde hacemos las pruebas deberíamos cambiarlo por un permitido/autorizado. En ese caso no podremos usar la creación fácil y deberemos usar la creación estándar (*Standard Create*). En la Figura 31 podemos ver los parámetros de configuración por defecto.



The screenshot shows a table titled "View default settings for Easy create". It lists various configuration parameters with their current values and whether they are editable after database creation. The columns are "Configuration", "Value", and "Editable after database is created".

Configuration	Value	Editable after database is created
Encryption	Enabled	No
VPC	Default VPC (vpc-7d90c807)	No
Option Group	default:mariadb-10-2	Yes
Subnet Group	default-vpc-7d90c807	Yes
Automatic Backups	Enabled	Yes
VPC Security Group	sg-56a22402	Yes
Publicly Accessible	No	Yes
Database Port	3306	Yes
DB Instance Identifier	OCT19-21GIIN	Yes
DB Parameter Group	default.mariadb10.2	Yes
Performance Insights	Enabled	Yes
Monitoring	Enabled	Yes
Maintenance	Auto Minor Version Upgrade Enabled	Yes
Delete Protection	Not Enabled	Yes

Figura 31. Paso 6 creación Base de Datos en AWS. Fuente: elaboración propia.

Una vez configurada, regresaremos a la pantalla de monitoreo y podremos ver que nuestra base de datos se está creando (*status: Creating*). Según la carga de los servidores de AWS en el momento en el que realicemos la creación, este proceso puede llevar varios minutos. En la Figura 32 podemos ver que al lado del texto nos aparece también un reloj indicando que AWS está trabajando en la creación de nuestra base de datos.

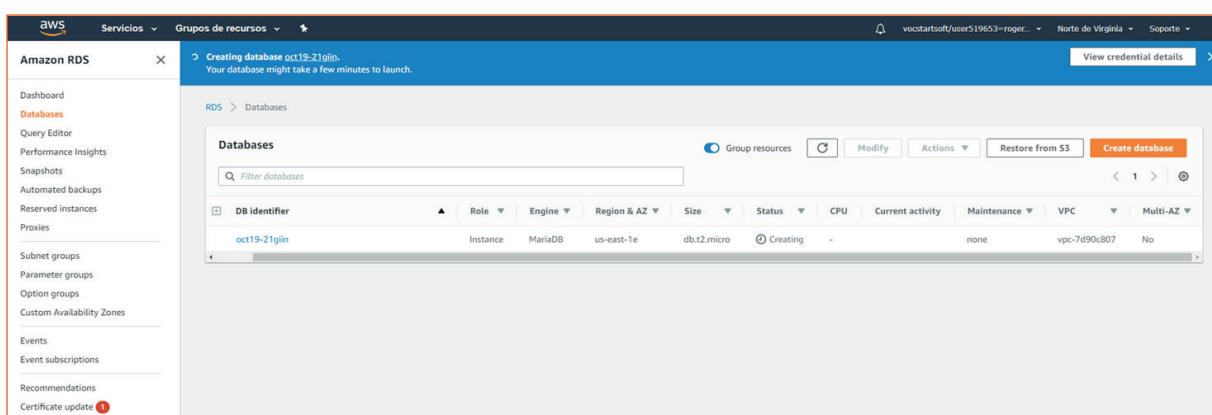


Figura 32. Paso 7 creación Base de Datos en AWS. Fuente: elaboración propia.

Una vez creada nuestra base de datos, haciendo clic en su identificador (*DB identifier*) podremos ver sus detalles. Es importante guardar la información sobre el *Endpoint* y el *Port*, ya que son datos imprescindibles para conectarse remotamente a la base de datos desde nuestro programa Java. El *Endpoint* corresponde al nombre/ip del servidor. En la Figura 33 podemos ver estos detalles, así como otros: uso de CPU, conexiones activas, información de red, etc.

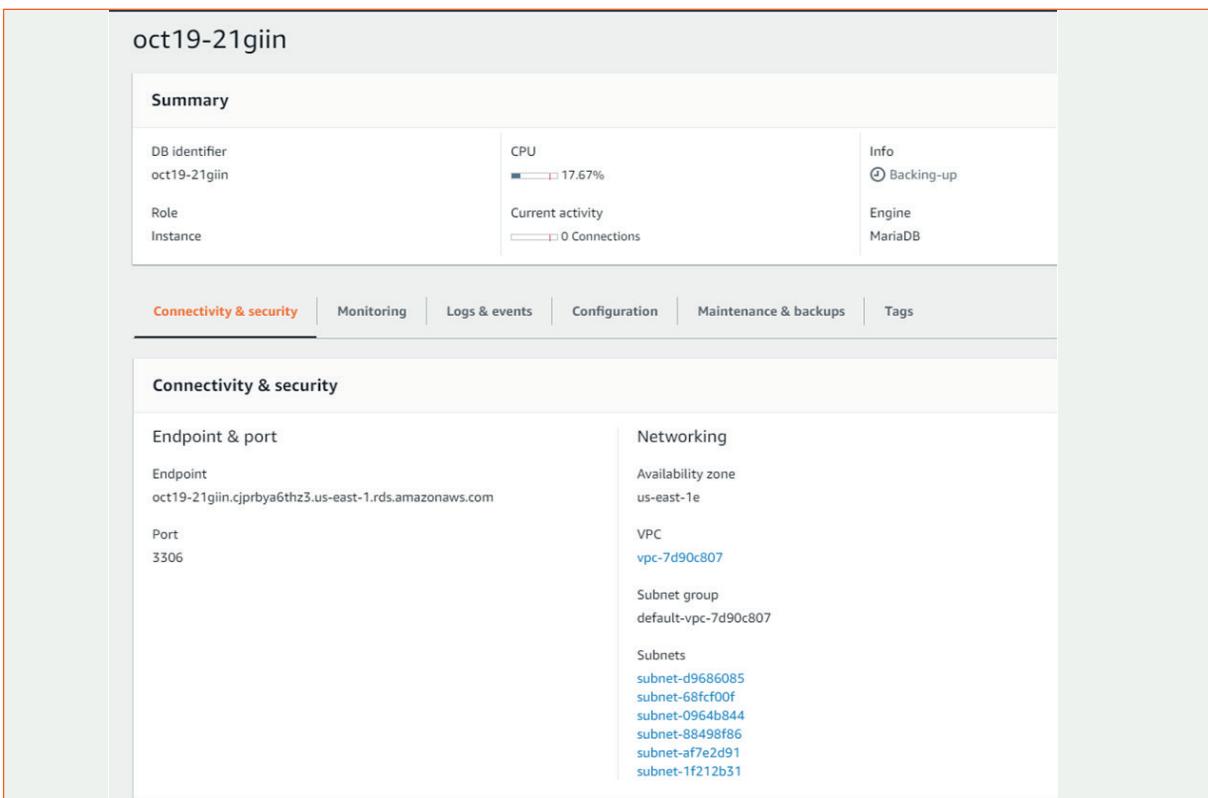


Figura 33. Paso 8 creación Base de Datos en AWS. Fuente: elaboración propia.

Aún no terminamos la configuración, para permitir el acceso remoto también deberemos cambiar algunos reglas de seguridad. Para hacer estos cambios debemos ir al grupo de seguridad (*Security Group*) hacer doble clic en él (Figura 34) y editar las reglas de entrada (*Edit inbound rules*) tal y como se muestra en la Figura 35.

Security group rules (2)		
Security group	Type	Rule
default (sg-56a22402)	EC2 Security Group - Inbound	sg-56a22402
default (sg-56a22402)	CIDR/IP - Outbound	0.0.0.0/0

Figura 34. Paso 9 creación Base de Datos en AWS. Fuente: elaboración propia.

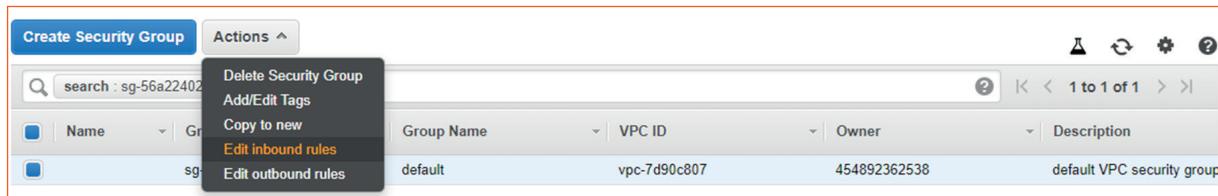


Figura 35. Paso 10 creación Base de Datos en AWS. Fuente: elaboración propia.

Es necesario agregar una regla que permita la conexión a la base de datos desde cualquier IP, siempre que dispongamos del usuario y la clave. Con 0.0.0.0 indicaremos cualquier IP. Si tuviésemos una IP estática en nuestro equipo podríamos indicarla. Tener una IP estática es la opción recomendada para bases de datos en producción para mejorar la seguridad (sólo podremos acceder desde nuestro servidor), pero para realizar el ejercicio no es tan importante. En la Figura 36 podemos ver los detalles de la regla a agregar para 0.0.0.0.

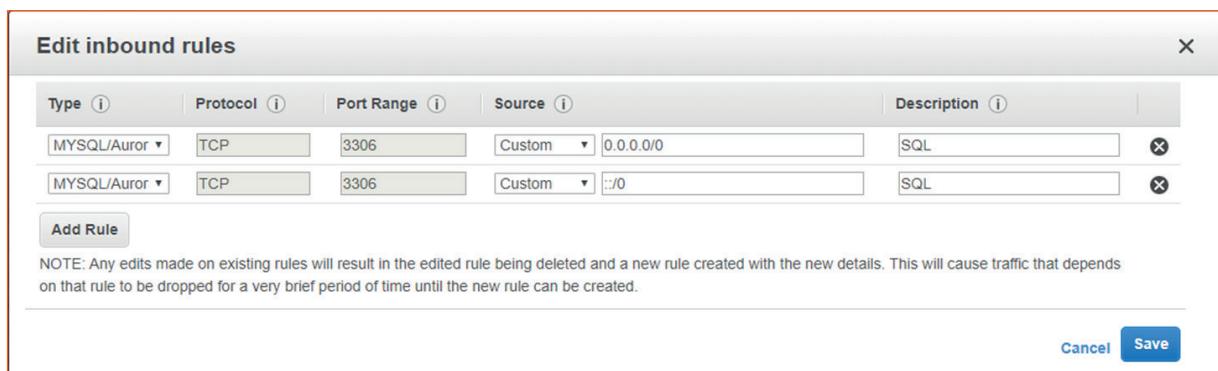


Figura 36. Paso 11 creación Base de Datos en AWS. Fuente: elaboración propia.

Por último, deberemos modificar la accesibilidad para que sea pública. Esto lo realizaremos con la opción modificar (*Modify*) (ver Figura 37), y seleccionando si (*yes*), ver Figura 38.

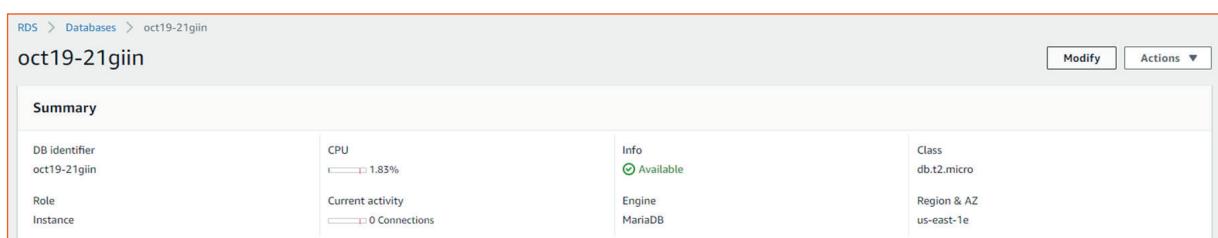


Figura 37. Paso 12 creación Base de Datos en AWS. Fuente: elaboración propia.

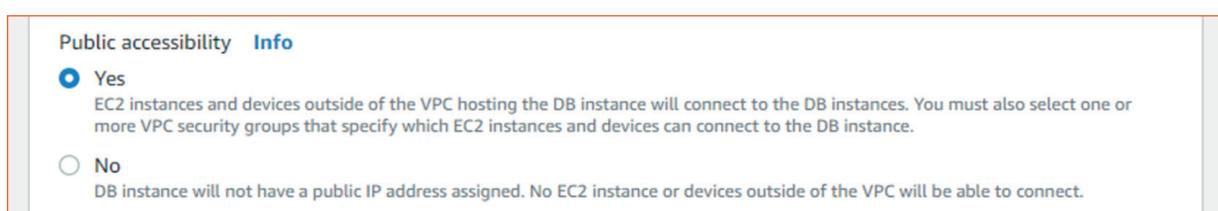


Figura 38. Paso 13 creación Base de Datos en AWS. Fuente: elaboración propia.

Una vez finalizada la creación y configuración es muy recomendable verificar que funciona correctamente, por ejemplo, utilizando el visualizador de bases de datos HeidiSQL, o cualquier otro que tenga una versión gratuita con la suficiente funcionalidad para esta verificación.



HeidiSQL

<https://www.heidisql.com/download.php>

8.2. Conector MySQL para Java

Para poder acceder desde nuestro código en Java a la base de datos, será necesario contar con un driver JDBC. Si nuestro entorno de desarrollo no lo tienen por defecto, podemos descargarlo de:



MySQL

<https://dev.mysql.com/downloads/connector/j/>

Normalmente con el .jar será la forma más fácil de agregarlo al entorno de desarrollo, seleccionaremos que el mismo sea independiente de la plataforma para facilitar que nuestro programa funcione sin problemas en cualquier equipo, manteniendo esta característica del Java. En la Figura 39, podemos ver los detalles para la descarga.

Figura 39. Descarga del conector JDBC. Fuente: elaboración propia.

Veamos una posible codificación de la conexión en Java, para realizar pruebas remplazar por el Endpoint asignado, puerto, usuario y claves correctos:

```
import java.sql.*;
Connection conn = null;
try {
    conn = DriverManager.getConnection("jdbc:mysql://oct19-21giin.cjprbya6tgc3.us-east-1.rds.amazonaws.com:3306/","admin","VIU");
    Statement stmt=conn.createStatement();
```

```

ResultSet rs=stmt.executeQuery("select * from prueba");
while(rs.next())
System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
conn.close();
}
catch(Exception e)
{ System.out.println(e);}

```

8.3. Código Java

Para entender el funcionamiento, primero veremos el diagrama de estados de nuestra calculadora. Tenemos tres estados posibles y según la interacción del usuario cambiaremos a otro estado o nos mantendremos en el actual.

Iniciamos siempre pidiendo un número que estará compuesto por n dígitos. En el momento en el que introduzcamos un operador verificaremos si este es uno de los operadores posibles, en ese caso pasaremos a pedir el segundo número, que estará compuesto por m dígitos y usaremos el *igual* para finalizar y volver al estado inicial. En la Figura 40 se puede ver de forma más clara.

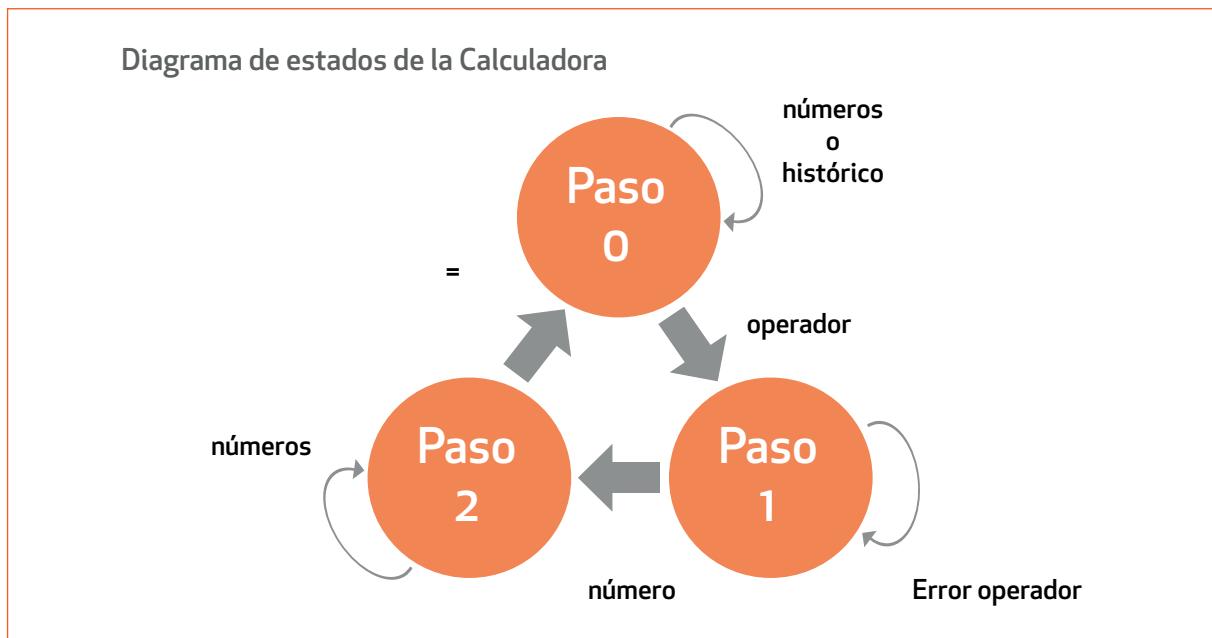


Figura 40. Diagrama de estados de la calculadora. Fuente: elaboración propia.

Otra opción es consultar el histórico de todas las operaciones realizadas. Para obtener este histórico el programa guarda en la base de datos cada operación realizada satisfactoriamente y esta opción realiza un consulta de todos los registros de la tabla guardados en la base de datos.

Y por último, en el caso que el operador no sea correcto (no exista), nos quedaremos pidiendo al usuario un nuevo operador valido. Por ejemplo, poner = antes de haber indicado los dos números y la operación a realizar.

A continuación, las dos clases que componen la calculadora:

CalcSQL

```
package ejcalcsql;

import java.awt.Frame;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class CalcSQL extends Frame implements WindowListener{
    Botones3 botoneraCalc;
    public CalcSQL()
    {
        super("CALCULADORA - Eventos + SQL");
        botoneraCalc=new Botones3();
        this.setSize(280, 125);
        this.add(botoneraCalc);
        this.addWindowListener(this);
    }
    public void windowActivated(WindowEvent event)
    {
        botoneraCalc.mensaje("Ventana activada.");
    }
    public void windowDeactivated(WindowEvent event)
    {
        botoneraCalc.mensaje("Ventana desactivada.");
    }
    public void windowIconified(WindowEvent event)
    {
        botoneraCalc.mensaje("Ventana minimizada.");
    }
    public void windowDeiconified(WindowEvent event)
    {
        botoneraCalc.mensaje("Ventana desminimizada.");
    }

    public void windowOpened(WindowEvent event)
    {
        botoneraCalc.mensaje("Ventana abierta.");
    }
    public void windowClosing(WindowEvent event)
    {
        botoneraCalc.mensaje("Ventana cerrandose.");
        this.dispose();
    }
}
```

```
}

public void windowClosed(WindowEvent event)
{
    botoneraCalc.mensaje("Ventana cerrada.");
    System.exit(0);
}
public static void main(String args[])
{
    CalcSQL app=new CalcSQL();
    app.setVisible(true);
}
}
```

Botones3

```
package ejcalcsql;
import java.awt.*;
import java.awt.event.*;
import java.sql.*;

public class Botones3 extends Panel implements ActionListener
{
    private TextField texto;
    private int num1 = 0;
    private int num2 = 0;
    private double resultado = 0;
    private String op = "";
    private int paso = 0;
    Connection conn = null;
    public Botones3()
    {
        LayoutManager layout=new FlowLayout();
        Button boton1=new Button("1");
        Button boton2=new Button("2");
        Button boton3=new Button("3");
        Button boton4=new Button("4");
        Button boton5=new Button("5");
        Button boton6=new Button("6");
        Button boton7=new Button("7");
        Button boton8=new Button("8");
        Button boton9=new Button("9");
        Button boton0=new Button("0");
        Button botonMas=new Button("+");
        Button botonMenos=new Button("-");
```

```

        Button botonDividir=new Button("/");
        Button botonMultiplicar=new Button("*");
        Button botonIgual=new Button("=");
        Button historicoSQL = new Button("Historico");
        texto=new TextField("Calculadora SQL");
        texto.setEditable(false);
        this.setLayout(layout);
        this.add(boton1);
        this.add(boton2);
        this.add(boton3);
        this.add(boton4);
        this.add(boton5);
        this.add(boton6);
        this.add(boton7);
        this.add(boton8);
        this.add(boton9);
        this.add(boton0);
        this.add(botonMas);
        this.add(botonMenos);
        this.add(botonDividir);
        this.add(botonMultiplicar);
        this.add(botonIgual);
        this.add(historicoSQL);
        this.add(texto);
        boton1.addActionListener(this);
        boton2.addActionListener(this);
        boton3.addActionListener(this);
        boton4.addActionListener(this);
        boton5.addActionListener(this);
        boton6.addActionListener(this);
        boton7.addActionListener(this);
        boton8.addActionListener(this);
        boton9.addActionListener(this);
        boton0.addActionListener(this);
        botonMas.addActionListener(this);
        botonMenos.addActionListener(this);
        botonDividir.addActionListener(this);
        botonMultiplicar.addActionListener(this);
        botonIgual.addActionListener(this);
        historicoSQL.addActionListener(this);
    }
    public void mensaje(String mensaje)
    {
        texto.setText(mensaje);
    }
}

```

```
        System.out.println(mensaje);
    }
    public void actionPerformed(ActionEvent event)
    {
        String accion=event.getActionCommand();
        if (paso == 0)
        {
            if ((accion != "+") && (accion != "-") && (accion != "/") &&
(accion != "*") && (accion != "=") && (accion != "Historico"))
            {
                num1 = num1 * 10 + Integer.parseInt(accion);
                texto.setText(Integer.toString(num1));
            }
            else
            {
                if (accion == "Historico")
                {
                    System.out.println("HISTORICO");
                    try //Print Datos BD
                    {
                        conn = DriverManager.getConnection("jdbc:mysql://
oct19-21giin.cjprbyi6thz3.us-east-1.rds.amazonaws.com:3306/21GIIN","admin",
"VIU");
                        Statement stmt=conn.createStatement();
                        ResultSet rs = stmt.executeQuery("select * from historico");
                        while(rs.next())
                            System.out.println(rs.getString("num1")+rs.
getString("op")+rs.getString("num2")+"="+rs.getString("resultado"));
                        conn.close();
                    }
                    catch(Exception e)
                    {
                        System.out.println(e);
                    }
                    System.out.println("-----");
                }
                else if ((accion != "="))
                {
                    paso = 1;
                    op = accion;
                    texto.setText(texto.getText().concat(op));
                }
                else
                {
```

```

        System.out.println("Error: Debe ser numero o operacion");
    }
}
}
else if (paso == 1)
{
    if ((accion != "+") && (accion != "-") && (accion != "/") &&
(accion != "*") && (accion != "=") && (accion != "Historico"))
    {
        paso = 2;
        num2 = Integer.parseInt(accion);
        texto.setText(texto.getText().concat(accion));
    }
    else
    {
        System.out.println("Error: Debe ser numero");
    }
}
else if (paso == 2)
{
    if ((accion != "+") && (accion != "-") && (accion != "/") &&
(accion != "*") && (accion != "=") && (accion != "Historico"))
    {
        num2 = num2 * 10 + Integer.parseInt(accion);
        texto.setText(texto.getText().concat(accion));
    }
    else if (accion == "=")
    {
        texto.setText(texto.getText().concat(accion).concat(resultado()));
        try //Guaradar en BD
        {
            conn = DriverManager.getConnection("jdbc:mysql://
oct19-21giin.cjprbya69hz3.us-east-1.rds.amazonaws.com:3306/21GIIN","admin",
"VIU");
            Statement stmt=conn.createStatement();
            stmt.executeUpdate("insert into historico (num1,num2,op,resultado)
values ("+num1+","+num2+","+op+","+resultado()+")");
            conn.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        paso = 0;
    }
}
}
```

```
        num1 = 0;
        num2 = 0;
        op = "";
    }
    else
    {
        System.out.println("Error: Debe ser numero o =");
    }
}
private String resultado()
{
    double res = 0;
    if (op == "+")
    {
        res = num1 + num2;
    }
    else if (op == "-")
    {
        res = num1 - num2;
    }
    else if (op == "/")
    {
        res = (double)num1 / (double)num2;
    }
    else if (op == "*")
    {
        res = num1 * num2;
    }
    else
    {
        System.out.println("Error: No hay definida una operacion");
    }
    return Double.toString(res);
}
}
```

La mayor parte del código se puede entender con lo visto en los diferentes temas de este manual. Deberemos modificar los strings de conexión con la base de datos por los que correspondan (dirección del servidor, puerto, usuario y clave).

Sólo hay dos puntos que no hemos hablado en el manual: las dos maneras de acceder a la base de datos MySQL des del código según queramos hacer una consulta (SELECT) o una modificación (INSERT, DELETE o UPDATE).

Para realizar consultas utilizamos:

```
conn = DriverManager.getConnection("jdbc:mysql://oct19-21giin.
cjprbra6thz3.us-east-1.rds.amazonaws.com:3306/21GIIN","admin","VIU");
Statement stmt=conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from historico");
while(rs.next())
System.out.println(rs.getString("num1")+rs.getString("op")+rs.
getString("num2")+"="+rs.getString("resultado"));
conn.close();
```

Para realizar modificaciones utilizamos:

```
conn = DriverManager.getConnection("jdbc:mysql://oct19-21giin.
cjprbya6tz3.us-east-1.rds.amazonaws.com:3306/21GIIN","admin","VIU");
Statement stmt=conn.createStatement();
stmt.executeUpdate("insert into historico (num1,num2,op,resultado)
values ("+num1+","+num2+", '"+op+"', "+resultado()+")");
conn.close();
```

Como podemos observar, la diferencia está en que usamos `executeQuery()` para las consultas, donde obtenemos como resultado un conjunto de resultados (las diferentes filas de las tablas que cumplan el criterio), los cuales podemos recorrer posteriormente uno a uno. O usamos `executeUpdate()` para las modificaciones, en este caso el resultado es un entero que nos indica el número de filas afectadas en la base de datos por la modificación.

La ejecución del programa mostrará una ventana simple, como podemos ver en la Figura 41, con los diferentes botones para digitar los número e indicar las operaciones.

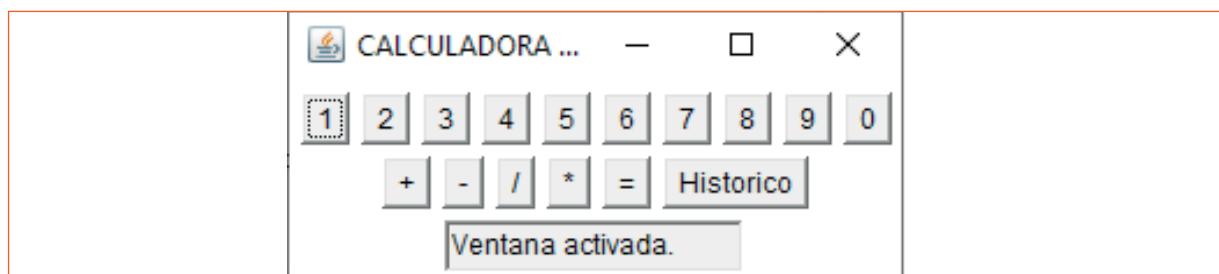


Figura 41. Calculadora. Fuente: elaboración propia.

A medida que vayamos introduciendo los datos de la operación, en el cuadrado de texto se va mostrando la misma, así como su resultado al pulsar =. Como podemos ver en la Figura 42.

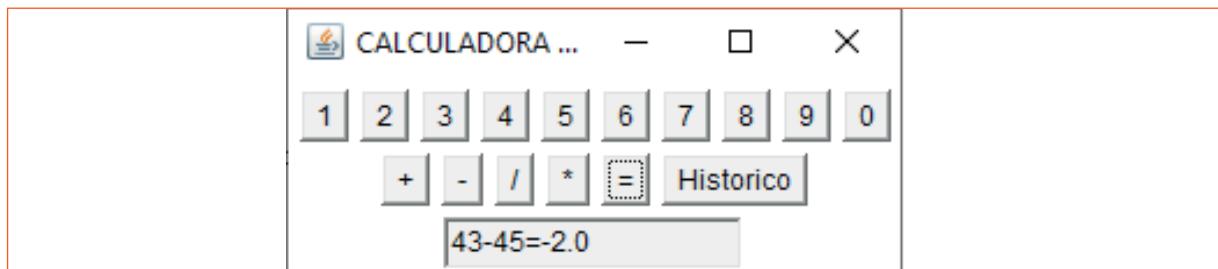


Figura 42. Calculadora mostrando el resultado de una operación. Fuente: elaboración propia.

Cuando pulsemos el botón "Historico" mostrará por consola, previa consulta con la base de datos, todas las operaciones realizadas.

Glosario

Amazon Web Services

Servicio en la nube de Amazon que proporciona diferentes servicios de computación bajo demanda de forma transparente mediante diferentes conjuntos de servidores (*clusters*) distribuidos por diferentes países.

Clase

Una clase es un tipo de datos más las funciones que actúan sobre el mismo. Adicionalmente cada clase será una abstracción de los objetos reales que queramos tener en nuestro sistema. Cuando asignamos valores a una clase y se convierte en un objeto "real" decimos que hemos hecho una instanciación de la clase.

Herencia

La herencia es el medio mediante el cual una clase obtiene todas las características y funcionalidades de otra clase, al mismo tiempo que nos permite extender dichas características y funcionalidades para hacerla más específica.

Instancia

Una instancia es cuando asignamos valores a una clase, para convertirla en un objeto "real" en nuestro contexto de trabajo.

Máquina Virtual Java

Emulación completa con: procesador, sistema operativo... implementada para cada plataforma que da soporte a Java. Este sistema virtual permite la portabilidad del código Java, pues realmente los programas no se compilan para un determinado entorno, sino para este entorno virtual que tienen múltiples implementaciones.

Objeto

Es un contenedor que integra tanto los datos, en forma de atributos; el código necesario para interactúa con los datos (o con una parte de ellos, pues podemos tener algunas restricciones de accesibilidad según como los definamos) y las acciones propias del objeto en forma de funciones/métodos. Algunos objetos pueden contener todos o algunos de los tres elementos nombrados, no es necesario tener todos ellos para que sean considerados objetos.

Programación Orientada a Objetos (POO)

Paradigma de programación basado en el concepto de objetos.

Unified Modeling Language (UML)

Estándar definido por *Object Management Group* que consiste en un conjunto de representaciones gráficas, o diagramas, que nos permiten definir de forma precisa los requerimientos de un programa.

Enlaces de interés

AWS Educat

Amazon Web Service Educate, enlace para crearse una cuenta de estudiante en AWS.

<https://aws.amazon.com/es/education/awseducate/>

Driver JDBC

Driver JDBC para utilizar bases de datos MySQL en Java.

<https://dev.mysql.com/downloads/connector/j/>

HeidiSQL

Visualizador de bases de datos HeidiSQL, tienen una versión gratuita.

<https://www.heidisql.com/download.php>

Java

Documentación de todas las clases estándares de Java proporcionada por Oracle. Específicamente este enlace es para la versión 8.

<https://docs.oracle.com/javase/8/docs/api/>

Swing

Tutorial de Swing, para implementar interfaces gráficas en Java.

<http://dis.um.es/~bmoros/Tutorial/introduccion/indice2.html#veintiuno>

Umbrello

Umbrello, programa en línea gratuito para la elaboración de diagramas UML

<https://umbrello.kde.org/>

UML

Web oficial de UML, donde encontraremos infinidad de recursos para su correcta utilización, así como la definición de los diferentes estándares.

<https://www.uml.org/>

Visual Paradigm

Visual Paradigm, programa en línea gratuito para la elaboración de diagramas UML.

<https://online.visual-paradigm.com/>

Bibliografía

Fernández, H. F. (2012). *Programación orientada a objetos con Java*. Colombia: Ecoe Ediciones.

Llinás, L. F. G. (2010). *Programación orienta a objetos en Java*. Colombia: Universidad del Norte.

López, E. T., Costa, D. C., & Samsó, M. R. S. (2004). *Especificación de sistemas software en UML*. Barcelona: Universitat Politècnica de Catalunya. Iniciativa Digital Politécnica.

Kimmel, P. (2007). *Manual de UML*. México: McGraw-Hill.

Agradecimientos

Autor

Dr. Roger Clotet Martínez



viu
.es