


Open Opened 2 weeks ago by  **Rubén Montero**

Comprendiendo el espacio de trabajo

Resumen

- Veremos una introducción breve a Java
- Comprenderemos cómo está montado el espacio de trabajo
- Hablaremos de las clases y métodos en Java
- Crearemos una clase `Greeter` con un método `sayHello`
- Probaremos el `test` automático de la tarea
- Distinguiremos `ejecutar Main.java` y `ejecutar tests`
- Haremos `git add`, `git commit` y `git push` para subir los cambios al repositorio

Descripción

[Java](#) es un lenguaje de programación [orientado a objetos](#). Nos otorga el poder de escribir el código fuente de programas que se [compilarán](#) y funcionarán en cualquier ordenador, independientemente de su arquitectura, con tal de que la [máquina virtual de Java \(JVM\)](#) esté disponible.

Se llama **código fuente** a cual programa o fragmento de programa que escribamos, ya sea en C, Python, Java,...

Primeros pasos

Para comenzar a trabajar con Java, lo primero será *bajar* este repositorio a tu disco duro.

Para ello necesitarás haber instalado [Git](#) y tener tu usuario, email y clave SSH configurados apropiadamente. Si seguiste estos pasos e hiciste `git clone`, ya tienes disponible el repositorio en local. Sólo necesitas abrir un terminal de Windows (*tecla de Windows* > `cmd`), posicionarte en tu repositorio local con `cd` y traer los cambios haciendo `pull`:

```
git pull
```

¡Hay una nueva carpeta en mi repositorio!

La carpeta `java-introduction` debe haber aparecido en la carpeta de tu repositorio local.

Abre [IntelliJ IDEA Community](#) y selecciona **File > Open**. Busca la carpeta `java-introduction` y elígela. Puedes darle a `Trust Project` sin problema. A la izquierda aparecerá la jerarquía de carpetas (Si no aparece automáticamente, haz *click* en la pequeña pestaña *Project*):

```
- .idea/

- assets/

- docs/

- lib/

- src/
  |
  |- Main.java

- tests/
```

- `.idea/` contiene información que IntelliJ IDEA guarda automáticamente (configuración del editor, etc.)
- `assets/` contiene ficheros que usaremos más adelante
- `docs/` contiene una imagen con un ejemplo. En general, es buena idea tener una carpeta separada para *documentación*
- `lib/` contiene librerías `.jar` con *bytecode Java* ya compilado y listo para la JVM. De momento, sólo hay 3 librerías. `junit` y `hamcrest` se necesitan para los `tests`. `org.json:json` sirve para manipular ficheros [JSON](#). Las usaremos más adelante.
- `src/` contiene **nuestro** código fuente, (ó *source code*). ¡Es la más importante! Única e irremplazable.
- `tests/` contienen *tests* automáticos para cada tarea. Cuando terminemos una tarea, podremos comprobar que está bien hecha lanzando el *test* automático asociado.

Pongámoslo en marcha

Como puedes observar, la clase `src/Main.java` ya tiene código escrito:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

...es el típico [Hello world](#) en Java. Podemos ejecutarlo, pero antes hay que añadir una *Configuración de ejecución*:

1. Dale a *Add Configuration...* en la barra superior, en el centro-derecha.
2. Selecciona *Add new...*, y en el menú desplegable, *Application* (el segundo elemento)
3. Puedes ponerle otro nombre (mejor **Main** en vez de **Unnamed**)
4. Hay que seleccionar *qué Main class* se ejecutará. Basta con teclear **Main** en el cuadro de texto resaltado, o bien seleccionar en el pequeño icono y darle a **OK** en la ventana que aparece.
5. Dale a **OK**

Ya aparecerá un **botón verde 'Play'**. Pulsémoslo. Alternativamente, podemos pulsar Mayus+F10. Ésto lanzará la *Configuración Main* que acabamos de crear, y, así, se ejecutará el método *estático* `main`¹.

¡Pruébalo!

Si se despliega un recuadro en la parte inferior y muestra algo como esto:

```
C:\Users\Developer\.jdk\openjdk-18.0.1.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition  
Hello world!  
  
Process finished with exit code 0
```

La ejecución ha sido correcta.

Programar... Con clase

La noción central de la programación orientada a objetos (POO) es la **clase**.

Una **clase** es un *tipo* de variable.

Tipos [primitivos](#)

En Java existen los siguientes *tipos* de variable [primitivos](#):

- **byte** : Almacena 1 byte de información (8 *ceros* ó *unos*). Numéricamente, permite almacenar un valor en el rango [-128, 127]
- **short** : Almacena un número entero en 2 bytes; [-32768, 32767]
- **int** : Almacena un número entero en 4 bytes; [-2³¹, 2³¹-1]
- **long** : Almacena un número entero en 8 bytes; [-2⁶³, 2⁶³-1]
- **float** : Almacena un número decimal² en 4 bytes. Es lo que se conoce como *precisión simple*
- **double** : Almacena un número decimal en 8 bytes. Precisión *doble*
- **boolean** : Almacena un valor que sólo puede ser **true** o **false**
- **char** : Almacena un carácter (una letra) en 2 bytes. Se emplea la codificación de caracteres UTF16.

Como ya sabes, se usan así:

```
int miNumero = 6;  
int otroNumero = 14;  
// Espero que este resultado sea 20!!  
int resultado = miNumero + otroNumero;
```

Tipos no tan primitivos

Pues, nosotros, creando una **nueva clase**, podemos definir un **tipo** de variable nuevo.

La nueva clase:

1. Debe ser escrita en un fichero nuevo (*De momento no trabajaremos con clases internas*)
2. Debe declararse con la palabra **class**, seguida del nombre de la clase y llaves ({ })
3. El nombre del fichero y el nombre de la clase deben ser iguales

Pruébalo.

Haz **click derecho** en la carpeta `src/` y selecciona **New > Java Class**. En el diálogo que aparece, escribe `Greeter` y dale a ENTER.

Un nuevo fichero `Greeter.java` se ha creado con el siguiente contenido³:

```
public class Greeter {  
}
```

¿Cómo se usa?

Igual que con los tipos primitivos, se **declara** una variable escribiendo el *tipo* y luego el *nombre de la variable*. Así:

```
int unNumero;  
float unNumeroDecimal;  
// El siguiente dato no es primitivo  
Greeter miVariableEspecial;
```

Cuando **declaramos** una variable, el sistema reserva la memoria RAM necesaria para almacenarla, pero **aún no tiene ningún valor**. (¿Cuánto vale `unNumero` en el ejemplo anterior?)

Para darle un valor, debemos **inicializarla**. Eso se consigue con la palabra reservada `new`, así:

```
int unNumero = 3;  
float unNumeroDecimal = 3.333;  
// El siguiente dato no es primitivo  
Greeter miVariableEspecial = new Greeter();
```

Recuerda que los nombres de las clases deben escribirse en UpperCamelCase (la primera letra mayúscula), y las variables en lowerCamelCase (la primera letra minúscula).

Pruébalo

En la clase `Main.java`, dentro del método `main`, **declara e inicializa** una nueva variable `Greeter` así:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
        Greeter greeter = new Greeter();  
    }  
}
```

Ya tengo mi variable... ¿Y ahora?

Dispones de todo el potencial que te otorgue la clase `Greeter`.

Por el momento, ese potencial es **absolutamente nada**.

Vamos a escribir **un método** en la clase `Greeter` para que sirva para algo.

¿Qué es un método?

Es un fragmento de código que se puede *invocar* (o *llamar*). Prácticamente lo mismo que una función o un procedimiento en programación estructurada.

Se escriben **dentro** de una clase (¡Ojo! No puedes escribir un método *dentro de otro método*, o en lugares raros).

Constan de las siguientes partes:

1. Visibilidad: `public`, `package`, `protected` ó `private`. Si no es específica, por defecto es `package`
2. Tipo de dato devuelto. Puede ser un dato primitivo (e.g.: `int`), una clase (e.g.: `Greeter`) o `void` si el método *no devuelve nada*
3. Nombre del método, escrito en lowerCamelCase.
4. Paréntesis indicando los argumentos del método. Si no recibe argumentos, se escriben paréntesis vacíos `()` igualmente.
5. Cuerpo del método entre llaves `{ }`

Todo esto ya lo sabes, aunque si hay alguna parte que te genera dudas, no te preocupes. Lo iremos viendo poco a poco.

Por el momento, escribe el siguiente método **dentro** de `Greeter.java`:

```
public class Greeter {  
    public void sayHello() { }
```

```
}
```

(¿Identificas cada una de las partes mencionadas anteriormente?)

Venga, haz algo

Este método **no hace nada**.

Como de momento sólo hemos visto cómo imprimir el mensaje **Hello world!** por consola, podemos hacer que nuestro método `sayHello()` haga eso mismo:

```
public class Greeter {
    public void sayHello() {
        System.out.println("Hello world!");
    }
}
```

Invocar el método

El método está **escrito** y está **dentro** de la clase, pero no va a pasar *nada* mientras no lo *invoguemos*

Recuerda que el punto de entrada de la aplicación es `Main.java`, así que desde ahí podemos invocarlo:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
        Greeter greeter = new Greeter();
        greeter.sayHello();
    }
}
```

Ejecútalo.

En la salida de ejecución verás que aparece **Hello world! 2 veces**.

Borraremos la siguiente línea para que sólo aparezca **1**:

```
public class Main {
    public static void main(String[] args) {
-        System.out.println("Hello world!");
        Greeter greeter = new Greeter();
        greeter.sayHello();
    }
}
```

Lanzar tests

Nuestro método de trabajo consistirá en ir modificando `Main.java` e ir creando clases para conseguir distintos objetivos.

Cuando terminemos, lanzaremos un *test* automático que validará nuestro trabajo.

Para esta primera tarea, *despliega* la carpeta `tests/` a la izquierda y elige `T001UnderstandingEnvironment`. Haz:

- Click derecho > **'Play' verde** Run 'T001UnderstandingEnvironment'

Se ejecutará el *test* automáticamente y en la ventana que se despliega abajo, a la izquierda verás *ticks* verdes e indicaciones de que el *test* ha sido exitoso.

Lanzar `Main` vs. lanzar tests

¡Ojo! Verás que arriba, donde le solíamos dar al **'Play' verde** para ejecutar `Main` **ya no pone** `Main`. Ahora indica `T001UnderstandingEnvironment`.

Si queremos volver a ejecutar `Main.java`, deberemos hacer *click* en el desplegable y seleccionar **Run configurations > Main**.

Por último

La tarea ha sido completada. Ahora, desde el terminal de Windows (cmd) escribiremos:

```
git add *
```




...para añadir los cambios a un nuevo *commit*. Después:

```
git commit -m "Completada tarea 1"
```

...y por último subimos los cambios al repositorio remoto:

```
git push
```

No está de más visitar la página de GitLab y verificar que el *commit* se ha subido.

1. ¿Te has preguntado por qué el método tiene `String[] args` ? 
2. Los números decimales se almacenan en posiciones de memoria que son *ceros* o *unos*. Esto se consigue mediante [IEEE754](#), y es importante recordar que en los ordenadores existen *error de precisión* con respecto a los números decimales o *en coma flotante* 
3. `public` es un modificador de visibilidad. Se escribe antes de nombres de clases, métodos o atributos, y sirve para indicar que *aquello a lo que modifica* debe ser *visible* (accesible) desde *cualquier otra parte del código del programa*. 



Rubén Montero @ruben.montero changed milestone to [%Sprint 1](#) 2 weeks ago



Ania Blanco @ania.blanco mentioned in commit [ce9aa885](#) 1 minute ago