

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                self.sck, client_addr = self.sck.accept()
                self.handle=self.sck
                self.todo=2
            elif self.todo==2:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            else:
                self.handle.send(data)
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

01.2

Introducción a Java

Indice

- Características
- Ediciones
- Instalación
- Edición y compilación
- Analizando el código
- Usando un IDE

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
    def listen(self,host,port,func):
        try:
            self.sock.bind((host,port))
            self.sock.listen(2)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sock.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            def send(self, data):
                self.handle.send(data)
            def close(self):
                self.flag=0
                self.sock.close()
```

Características (I)

Java es uno de los lenguajes de programación más utilizados, especialmente en el ámbito empresarial, en lo que se refiere a sistemas distribuidos de gestión y servicios.

Los principales **objetivos de diseño** de Java fueron⁽¹⁾:

- **Simple, Orientado a Objetos y Familiar**
 - Simple, para facilitar su aprendizaje y favorecer la productividad.
 - Es un lenguaje orientado a objetos, al entender dicho paradigma como la mejor opción en entornos altamente distribuidos e interconectados.
 - Al mismo tiempo, se adopta una sintaxis similar a la de C++ para hacerlo familiar a los programadores y facilitar su migración al nuevo lenguaje

⁽¹⁾ <https://www.oracle.com/technetwork/java/intro-141325.html>

Características (II)

- Robusto y seguro
 - Diseñado para que las aplicaciones sean altamente fiables. De **tipado estático**, se realizan chequeos del código en tiempo de compilación y ejecución.
 - Gestión dinámica de la memoria. El programador determina cuándo se crean los objetos pero es el entorno, es decir, la JVM, la que se encarga de la gestión del ciclo de vida del mismo: asignación automática de memoria a los nuevos objetos creados y recuperación de recursos una vez destruidos (*garbage collector*).
 - Dados los problemas de seguridad que conlleva operar en entornos distribuidos, Java define un modelo para controlar y limitar el acceso a los recursos desde los programas y aplicaciones (*sandbox*).

Características (III)

- **Independiente de la Arquitectura y Portable**

Java se diseñó para entornos distribuidos y heterogéneos, donde coexiste gran variedad de plataformas hardware y sistemas operativos. Al no compilar directamente a código máquina sino a un **bytecode** que ejecutará una máquina virtual de Java, se garantiza la portabilidad del código generado a cualquier plataforma que disponga de una implementación dicha máquina virtual

- **Alto rendimiento**

Si bien inicialmente la velocidad de ejecución de las aplicaciones era pobre, a lo largo de sus diferentes versiones se ha realizado un esfuerzo considerable. El interfaz JNI para la ejecución de código nativo y, especialmente, la introducción de **compiladores JIT** (*Just-In-Time*) en tiempo de ejecución supusieron grandes avances

Ediciones (I)

Java Standard Edition (JSE)

- Contiene las librerías y herramientas necesarias para desarrollar aplicaciones de escritorio y servidor.
- Java ha sufrido numerosas transformaciones desde la liberación del JDK 1.0 (*Java Development Kit*) el 23 de enero de 1996. La primera versión estable, JDK 1.0.2, se denominó Java 1.
- Desde J2SE 1.4, la evolución del lenguaje ha sido regulada por el JCP (*Java Community Process*), que usa Java Specification Requests (*JSRs*) para proponer y especificar cambios en la plataforma Java
- Desde J2SE 1.5 las versiones empezaron a denominarse J2SE 5 (y sucesivos)
- La última versión es J2SE 15, publicada en septiembre de 2020

Ediciones (II)

Version	Release date	End of Free Public Updates ^{[1][4]}	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	January 2019 for Oracle (commercial) December 2020 for Oracle (personal use) At least May 2026 for AdoptOpenJDK At least May 2026 ^[5] for Amazon Corretto	December 2030
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	At least October 2024 for AdoptOpenJDK At least September 2027 ^[5] for Amazon Corretto	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA
Legend: ■ Old version ■ Older version, still maintained ■ Latest version ■ Future release			

- Obsoleta
- Long Term Support
- Última

Ediciones (III)

Una nota sobre las licencias

- Con la publicación de J2SE 11, Oracle modificó la licencia de Java, dejando de ser gratuito para uso comercial (2,5\$ mes/usuario de escritorio ó 25\$ mes/procesador para aplicaciones de servidor) a cambio de ofrecer soporte extendido. No se aplica a versiones anteriores de Java
- Paralelamente, se ha adoptado **OpenJDK** como implementación de referencia de Java SE y **libre** para uso comercial, así como repositorio del código fuente de Java. Incluye los binarios mantenidos por Oracle pero, Oracle sólo proporcionará actualizaciones (p.e de seguridad) para la última versión activa
- **AdoptOpenJDK** es una iniciativa que proporciona los binarios de Java basados en OpenJDK pero permite seleccionar entre las VM HotSpot de OpenJDK y OpenJ9 de Eclipse. Mantenido por la comunidad

Ediciones (y IV)

Otras Ediciones

Java Enterprise Edition (Java EE)

- Plataforma para el desarrollo y ejecución de aplicaciones Java de red distribuidas en capas y gran escala.
- Se apoya en componentes de software modulares ejecutándose sobre **servidores de aplicaciones**

Java Micro Edition (Java ME)

- Especificación de un subconjunto de la plataforma Java orientada al desarrollo para dispositivos de recursos limitados (PDA, móviles, electrodomésticos,...)
- Si bien tuvo cierta relevancia en el desarrollo de juegos y apps para teléfonos móviles (pre-smartphone) hoy está en desuso

Instalación (I)

- Para poder trabajar con Java, necesitaremos tener instalado en nuestro equipo el J2SE en alguna de sus versiones (en nuestro caso, J2SE 8)
- Dependiendo de nuestras necesidades, podremos instalar uno de los siguientes paquetes:
 - **JRE (Java Runtime Environment)**. Contiene lo necesario para la ejecución de aplicaciones Java en nuestro equipo (máquina virtual Java, librerías,...)
 - **Server JRE**. Versión de JRE específica para desplegar aplicaciones Java en servidores. Incluye herramientas de monitorización de la JVM y otras de uso común en servidores
 - **JDK (Java SE Development Kit)**. Concebido para desarrolladores. Incluye un JRE junto con las herramientas necesarias para desarrollar, compilar, depurar y monitorizar aplicaciones Java

Instalación (II)

- En nuestro caso, usaremos el JDK de Java SE 11, última versión LTS
- Podemos optar tanto por la versión de Oracle como por la de OpenJDK
- En el sitio web de Oracle para Java se encuentran los archivos de instalación e instrucciones correspondientes, tanto para Windows como para Linux y Mac OS X:
 - <https://www.oracle.com/java/technologies/javase-downloads.html>
- Los instalables de JDK del OpenJDK los podemos encontrar en:
 - <https://jdk.java.net/>
- A modo de ejemplo, la instalación de OpenJDK 11 desde repositorios en Ubuntu 20.04 la podríamos realizar desde una consola ejecutando los siguientes comandos:

```
bowman@hal:~$ sudo apt update
bowman@hal:~$ sudo apt openjdk-11-jdk
```

Instalación (III)

Variables de Entorno

- Para poder trabajar adecuadamente con Java, es necesario tener configuradas algunas variables de entorno del sistema:

PATH

- Establece las rutas a las carpetas en las que se va a buscar cualquier comando que lancemos desde la consola (p.e., el compilador de Java). De no configurarse, tendremos que especificar la ruta completa al ejecutable cada vez que lo invoquemos, como en:

```
[win] C:\> "C:\Program Files\Java\jdk1.8.0\bin\javac" MyClass.java  
[linux] ~$ /usr/lib/jvm/java-8-oracle/bin/javac MyClass.java
```

Se configurará para añadir la **ruta completa** al directorio **bin** del JDK, de forma que podremos invocar el comando directamente sin la ruta:

```
[win] C:\> javac MyClass.java  
[linux] ~$ javac MyClass.java
```

Instalación (IV)

JAVA_HOME

- Es usada por aplicaciones de terceros basadas en Java para encontrar la ubicación (ruta) del JDK (ó JRE) que necesitan para ejecutarse

Se configura para añadir la **ruta completa** al directorio *de instalación* del JDK (ó JRE)

CLASSPATH (alternativamente, opción -cp del compilador javac)

- Empleada por la JVM para encontrar las clases necesarias para ejecutar una aplicación Java (p.e., librerías externas de terceros, programas,...). Su valor por defecto es “.”, es decir, nuestro directorio actual

No necesitamos modificarla. De hacerlo, al **sobrescribirse** el valor por defecto, necesitamos añadir el directorio actual “.” al nuevo *classpath*

Instalación (y V)

Verificación

- Una vez instalado el JDK, podemos verificar que se ha instalado y configurado correctamente simplemente invocando la máquina virtual desde la línea de comandos:

```
bowman@hal:~$ java -version
openjdk version "11.0.8" 2020-07-14
OpenJDK Runtime Environment (build 11.0.8+10-post-Ubuntu-0ubuntu120.04)
OpenJDK 64-Bit Server VM (build 11.0.8+10-post-Ubuntu-0ubuntu120.04, mixed mode, sharing)
```

La opción **-version** del comando **java** solicita que se nos informe de las versiones utilizadas de máquina virtual y del entorno de ejecución (en este caso, la versión 11.0.8 de OpenJDK)

De recibir un mensaje del tipo “**comando no encontrado**”, deberíamos revisar la variable de entorno PATH para verificar que está convenientemente configurada

Edición y compilación en Java (I)

1) *Edición*

- Vamos a editar y compilar un pequeño programa de Java. Para ello, utilizaremos un simple editor de texto para añadir las siguientes líneas a un archivo denominado **Hola.java**

```
/**
 * Hola.java
 * Programa que imprime un mensaje de saludo
 *
 * @author Bowman
 */
public class Hola {
    /* la ejecución de la aplicación
       se inicia en el método main */
    public static void main(String[] args) {
        System.out.println("Hola, Mundo Java!"); // muestra el mensaje en la consola
    } // fin del método main
} // fin de la clase Hola
```

Edición y compilación en Java (II)

2) Compilación

- Una vez creado nuestro archivo con el código fuente, el siguiente paso será emplear el **compilador** de Java (**javac**) para generar el archivo con el código objeto (**bytecode**) que ejecutará la VM de Java
- En una consola, desde la misma carpeta en que tenemos el archivo **Hola.java** con el código fuente, ejecutaremos:

```
bowman@hal:~/work/src/java$ javac Hola.java
```

- Si la compilación se realizó sin errores, observaremos que, en la misma carpeta, hay un nuevo archivo con el mismo nombre (*Hola*) pero extensión **.class**. Este será el archivo compilado que contiene el **bytecode**

```
bowman@hal:~/work/src/java$ ls Hola.*  
-rw-rw-r-- 1 zeroth zeroth 420 jul 25 13:33 Hola.class  
-rw-rw-r-- 1 zeroth zeroth 294 jul 25 13:24 Hola.java
```

Edición y compilación en Java (y III)

3) *Ejecución*

- Finalmente, ya podemos ejecutar nuestro pequeño programa. Para ello, desde la misma carpeta en que tenemos el nuevo archivo **Hola.class**, lanzaremos la Java VM con el comando `java` y le pasaremos el **nombre** del archivo que contiene el bytecode (**sin la extensión .class**). En realidad, lo que le estamos indicando, es el **nombre de la clase** que queremos ejecutar

```
bowman@hal:~/work/src/java$ java Hola  
Hola, Mundo Java!
```

- Si todo fue correcto, debería mostrarse en la consola el mensaje “**Hola, Mundo Java!**”
- Hemos editado, compilado y ejecutado nuestro primer programa Java. Veamos ahora más detalladamente cada uno de los pasos del proceso

Analizando el código (I)

Comentarios en los programas

- Insertamos comentarios con objeto de **documentar** y mejorar la **legibilidad** del código fuente. Por ejemplo:

```
/**  
 * Hola.java  
 * Programa que imprime un mensaje de saludo  
 ...
```

- El compilador **ignora** dichos comentarios. Sólo tiene una función **informativa** para los propios desarrolladores
- En Java existen tres tipos de comentarios:
 - Comentarios de **fin de línea //**
 - Comentarios **tradicionales /* */**
 - Comentarios **Javadoc /** */**

Analizando el código (II)

Comentarios de fin de línea

- Se indican con el símbolo //
- Todo el texto que va a continuación, hasta el **fin de la línea**, es considerado por el compilador como un comentario y será **ignorado**

```
public static void main(String[] args) {  
    System.out.println("Hola, Mundo Java!"); // muestra el mensaje en la consola
```

Comentarios tradicionales

- Comienzan y termina con los delimitadores /* y */
- Pueden ocupar varias líneas
- El compilador ignora todo el texto entre los delimitadores

```
/* la ejecución de la aplicación  
   se inicia en el método main */  
public static void main(String[] args) {
```

Analizando el código (III)

Comentarios de Javadoc

- Comienzan y termina con los delimitadores `/**` y `*/`
- Pueden ocupar varias líneas
- El compilador ignora todo el texto entre los delimitadores
- La utilidad `javadoc` genera documentación de las clases en formato HTML a partir de ellos
- Se pueden incluir etiquetas (*tags*) predefinidas, que empiezan con el símbolo `@`, que tienen un significado concreto. Por ejemplo: `@author`, `@version`, `@param`, `@return`, `@throws`,...

```
/**
 * Hola.java
 * Programa que imprime un mensaje de saludo
 *
 * @author Bowman
 * @version 1.0
 */
```


Analizando el código (IV)

Declarando una clase

- La siguiente línea declara la clase **Hola**

```
public class Hola
```

- Todo programa en Java está formado por, al menos, una clase. Todo el código de nuestros programas estará contenido dentro de esta(s) clase(s)
- Para declarar una clase utilizamos la **palabra reservada *class*** que ira seguida por el **nombre de la clase** (en este caso, Hola)
- En este caso, la declaración de la clase incluye la **palabra reservada *public***. Es un **modificador de acceso**, que indica que dicha clase es de acceso público (volveremos sobre ello más adelante)
- Toda clase ***public*** debe almacenarse en un archivo del **mismo nombre** y extensión **.java**

Analizando el código (V)


Nombres de clases e identificadores

- El nombre de una clase, al igual que de las variables, métodos o atributos, se denominan **identificadores**
- Un **identificador** es una secuencia de caracteres que **sólo** pueden ser letras, dígitos, guiones bajos (`_`) y el signo `$`. Además, el primer carácter no puede ser un dígito
- Java **distingue** entre mayúsculas y minúsculas. Por ejemplo, no es lo mismo **Hola** que **hola**
- Por convención, los nombres de las clases comienzan con una letra mayúscula. Cuando el nombre de la clase está formado por varias palabras, se emplea la notación **Camel Case** (*UpperCamelCase*), de forma que la primera letra de cada palabra se pone en mayúsculas. Por ej:
 - ClaseDeEjemplo, HolaMundo, CuentaBancaria, ...

Analizando el código (VI)

Cuerpo de la clase

- El código fuente contenido por la clase se encierra entre **llaves {}**





```
public class Hola {   
    /* la ejecución de la aplicación  
       se inicia en el método main */  
    public static void main(String[] args) {  
        System.out.println("Hola, Mundo Java!"); // muestra el mensaje en la consola  
    } // fin del método main  
 } // fin de la clase Hola
```

- Java hereda gran parte de su sintaxis de los lenguajes C y C++, donde las llaves se emplean para delimitar **bloques de código**, es decir, “porciones” del código fuente que pertenecen a alguna unidad funcional, como puede ser: una clase, un método, una sentencia condicional o iterativa,...

Analizando el código (VII)

Bloques y sangrado

- Para facilitar la lectura del código, dentro de cada bloque delimitado por las llaves, se establece un “nivel” de **sangrado**, de forma que todo el código que forma parte del mismo bloque se encuentra “alineado”.
- Cada vez que se inicia un nuevo bloque, se incrementa el sangrado
- Se suelen emplear **dos** o **cuatro** espacios para el sangrado
- Los editores de los IDE suelen hacerlo de forma automática al detectar un nuevo bloque por la introducción de las llaves

```
public class Hola {  inicio bloque de la clase  
    /* la ejecución de la aplicación  
       se inicia en el método main */  
    public static void main(String[] args) {  inicio bloque del método  
        System.out.println("Hola, Mundo Java!"); // muestra el mensaje en la consola  
 } // fin del método main  
 } // fin de la clase Hola
```

Analizando el código (VIII)

El método *main*

- La línea:

```
public static void main(String[] args) {  
    . . .  
} // fin del método main
```

declara un bloque de construcción del programa denominado **método**.

- Los métodos, que en lenguajes procedimentales se denominan **procedimientos** o **funciones**, nos permiten agrupar en una única unidad identificada por un **nombre** (en este caso *main*), un conjunto de sentencias o instrucciones para realizar una tarea determinada.
- Una vez definido, podremos invocar al método utilizando el nombre del mismo, de forma que se ejecuten las instrucciones que contiene
- Las clases de Java, por lo general, contendrán uno o más métodos

Analizando el código (IX)

- El método *main* es un método con un significado especial, pues es la línea a partir de la cuál la JVM inicia la ejecución de nuestro programa.
- En una aplicación Java, **sólo uno** de los métodos debe denominarse *main* y debe declararse exactamente con la sintaxis:

```
public static void main(String[] args) {
```

- Aunque a lo largo del curso iremos viendo en detalle el significado de cada uno de los “términos” de la declaración, brevemente:
 - *public* y *static*, son modificadores de acceso al método (quién y cómo lo puede invocar)
 - *void*, indica que dicho método no devuelve ningún resultado
 - *String[] args*. Entre paréntesis se encierran los parámetros (argumentos) necesarios por dicho método. En este caso, sería una “lista” de valores de tipo *cadena de caracteres* (String)

Analizando el código (X)

Salida por la pantalla (consola)

- La línea:

```
System.out.println("Hola, Mundo Java!");
```

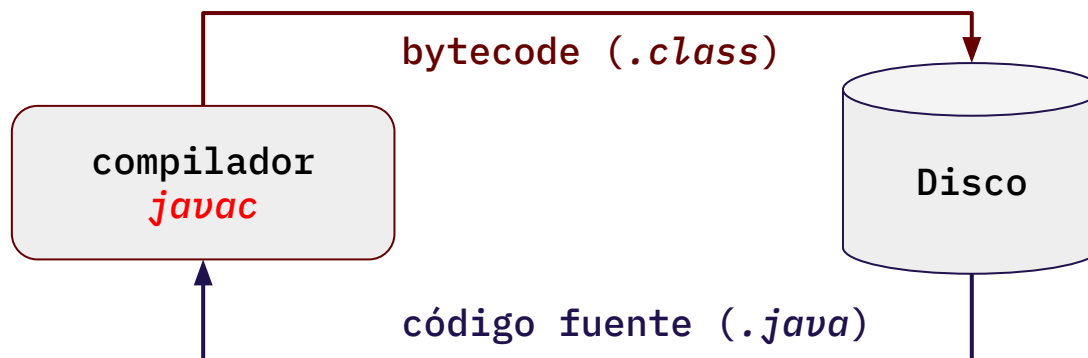
provoca que el mensaje “Hola, Mundo Java!”, se muestre en la pantalla

- La salida es realizada por el método *println*, que provoca que se imprima en pantalla el texto pasado como argumento
- Este método, pertenece al objeto *out*, que está conectado con la salida estándar del sistema
- A su vez, *out*, pertenece a la clase predefinida *System*, que proporciona acceso a diferentes recursos del sistema
- Al ser una sentencia completa, la línea termina con ;
- **Todas las sentencias** en Java terminan en ;

Analizando el código (XI)

Compilación. Errores

- Durante el proceso de compilación del código fuente, es decir, la generación del **código de bytes** (*bytecode*) que entiende la máquina virtual de Java (JVM), se realiza un análisis léxico, sintáctico y semántico de nuestro código fuente, con objeto de verificar que se ajusta a las especificaciones de Java.
- El **archivo binario** resultado de dicha compilación, tendrá el mismo nombre del archivo fuente (.java) y extensión **.class**



El archivo *.class* con el *bytecode* es **portable**. Lo podemos ejecutar en cualquier dispositivo que disponga de una JVM

Analizando el código (XII)

- Por lo general, los IDE, dispondrán de opciones en su menú (como *Build* o *Make*) que invocaran el comando *javac* por nosotros
- En caso de que se produjeran **errores** durante la compilación, *javac* nos indicará el tipo de error correspondiente y, de ser el caso, la línea de código en que se produjo.
- Por ejemplo, supongamos que eliminamos el símbolo (;) de la línea:

```
System.out.println("Hola, Mundo Java!");
```

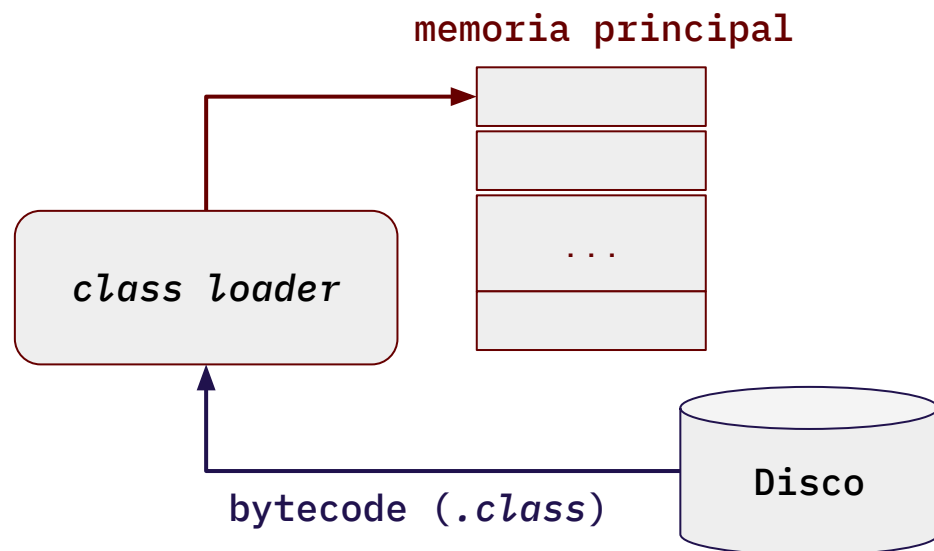
- Al compilar, *javac* nos informaría del error de sintaxis correspondiente y se abortaría la compilación

```
bowman@hal:~/work/src/java$ javac Hola.java
Hola.java:9: error: ';' expected
    System.out.println("Hola, Mundo Java!") // muestra el mensaje en la consola
                                   ^
1 error
```

Analizando el código (XIII)

Ejecución

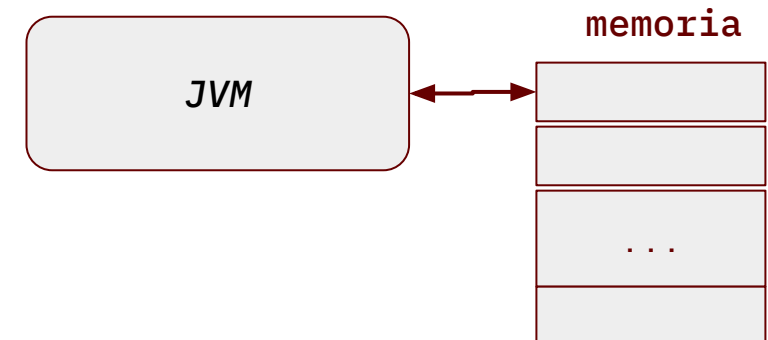
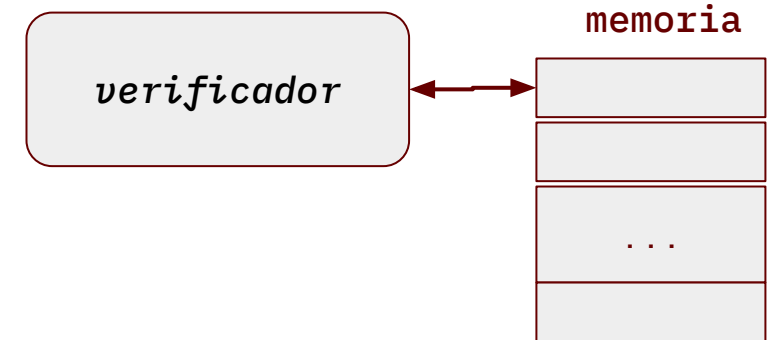
- El comando *java* ejecuta una aplicación Java. Para ello, inicia el entorno de ejecución de Java (JRE), carga la clase especificada como argumento e invoca el método *main* de dicha clase.
- En primer lugar, el *cargador de clases* de la JVM, extrae los *bytecode* de los archivos *.class* y los transfiere a la memoria principal



El *class loader* cargaría automáticamente cualquier otro archivo *.class* con clases adicionales utilizadas por la aplicación. Las clases podrían cargarse desde otros orígenes además del disco local: red, Internet,...

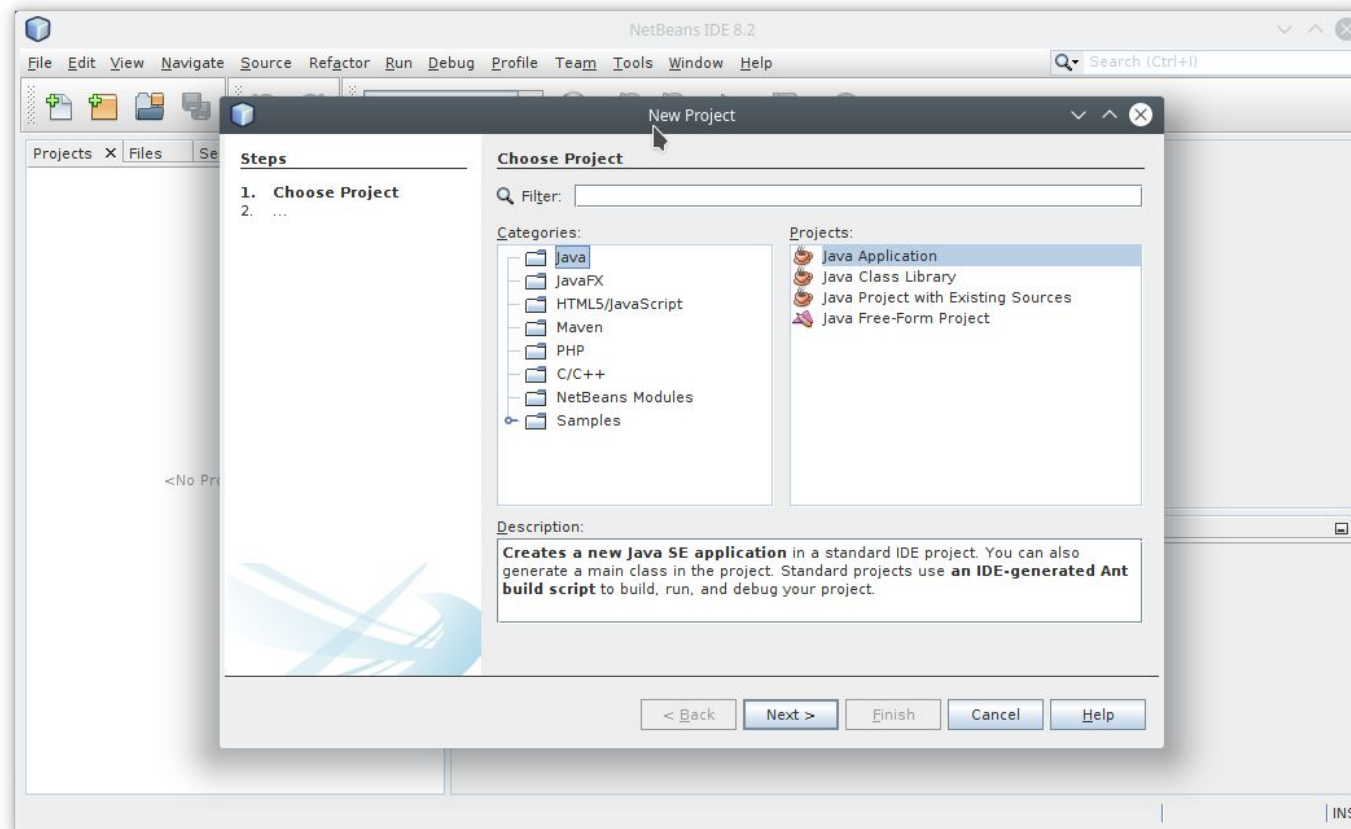
Analizando el código (y XIV)

- A medida que se cargan las clases, el *verificador de bytecode* examina los códigos de bytes para asegurar que sean válidos y que no violen las restricciones de seguridad de Java
- Finalmente, la JVM ejecutará el *bytecode*. Originalmente se trataba de un *intérprete* que *traducía* y ejecutaba cada una de las sentencias, siendo el proceso lento. Actualmente, se emplean compiladores **JIT** que compilan a código objeto secciones que se ejecutan con frecuencia (*active points*) mejorando el rendimiento



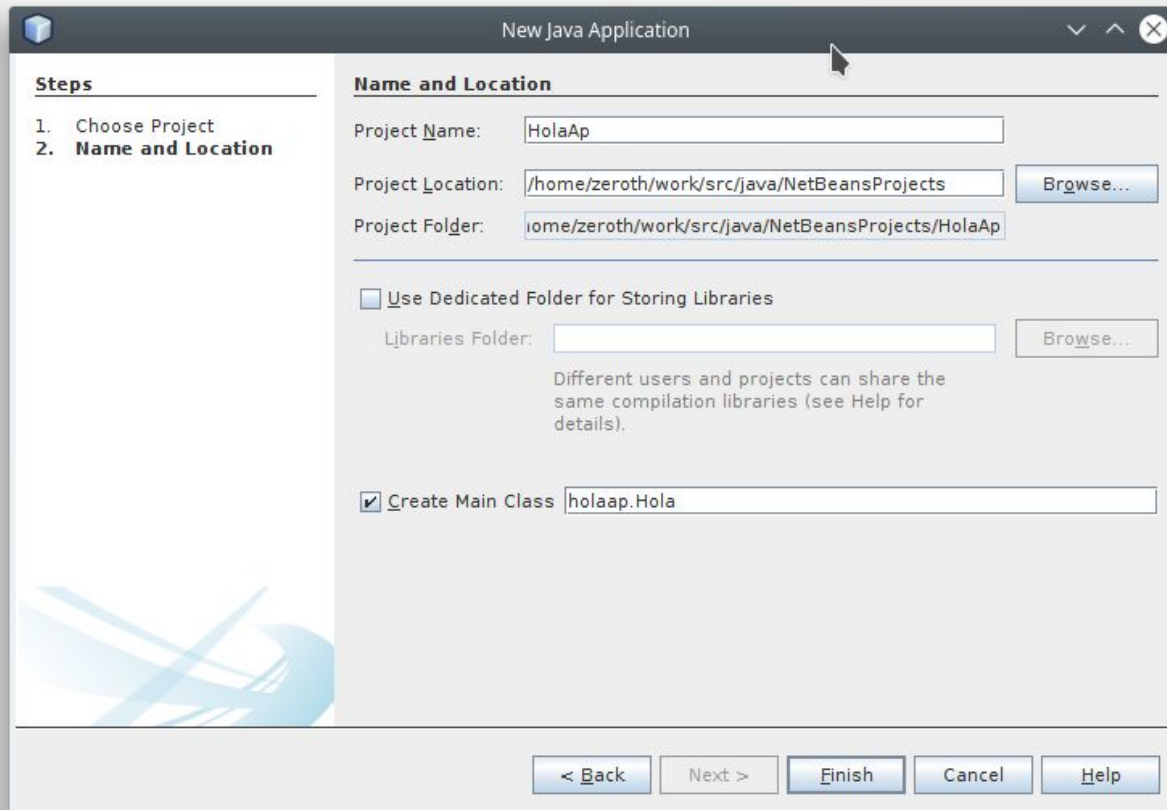
Usando un IDE (I)

- Repitamos el ejemplo anterior pero desde un IDE como NetBeans
- Una vez iniciado, seleccionaremos: *File > New Project*
En el diálogo que se muestra a continuación, seleccionaremos el tipo de proyecto *Java Application* de la categoría *Java*



Usando un IDE (II)

- Tras pulsar en *Next*, deberemos cubrir los datos principales del nuevo proyecto:

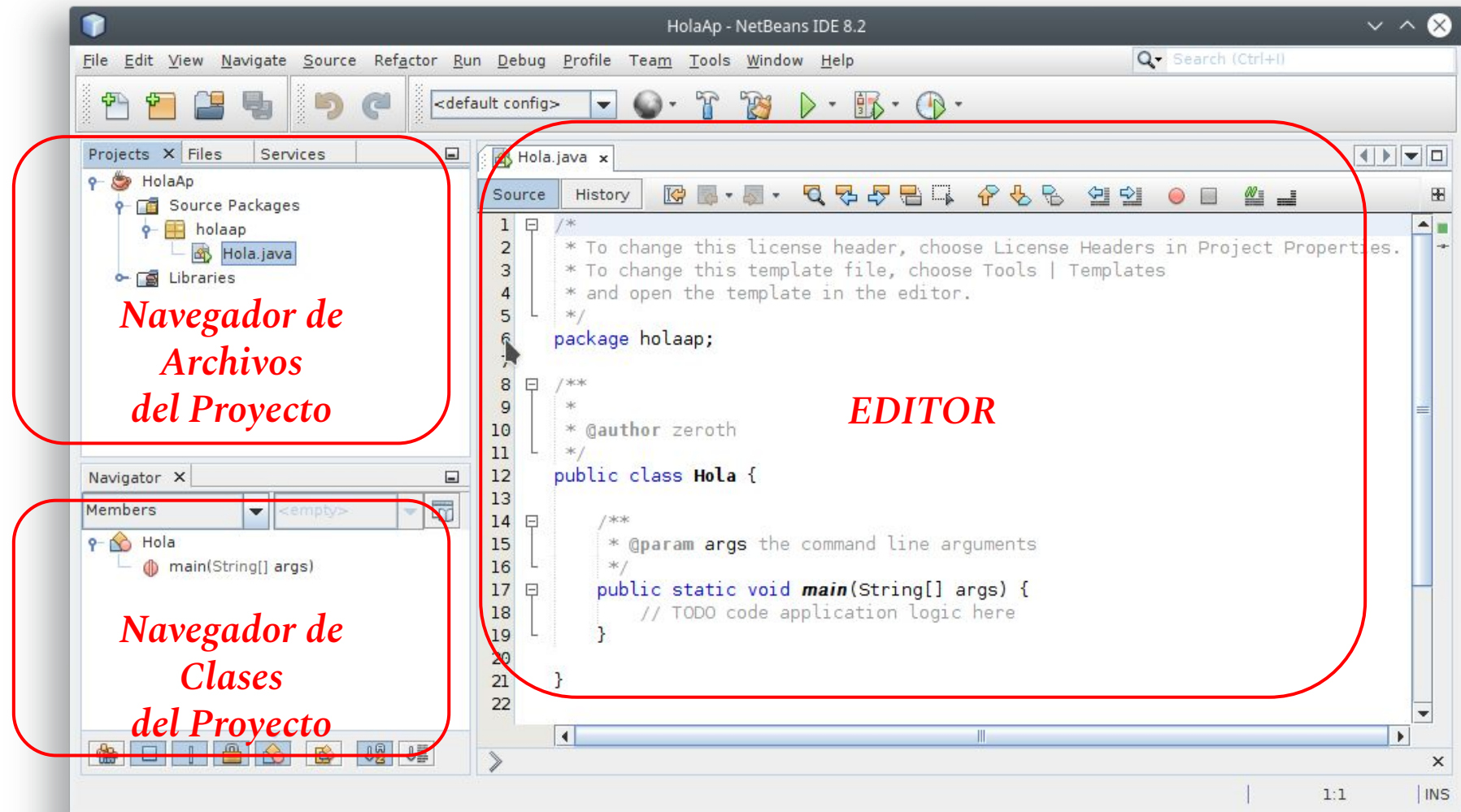


- Project Name: **HolaAp**
(*nombre del proyecto*)
- Project Location/Folder
(*ubicación archivos proyecto, podemos dejar los valores por defecto*)
- Main Class: **holaap.Hola**
(*clase principal. contiene el método main*)

- Y pulsamos sobre *Finish* para completar la creación del proyecto

Usando un IDE (III)

- Observamos que NetBeans ha creado un nuevo proyecto que contiene el archivo *Hola.java* con la estructura básica de la clase *Hola*



Usando un IDE (IV)

- Observa como NetBeans ha añadido una línea nueva al principio del archivo *Hola.java*:


```
package holaap;
```
- Indica que la clase pertenece al **paquete** *holaap*
- Los paquetes Java son unidades organizativas, similares a las carpetas de archivos, que nos permiten agrupar y organizar nuestras clases. Por defecto, NetBeans crea un nuevo paquete con el nombre del proyecto
- Cuando no se indica un paquete, Java asigna dichas clases al **paquete por defecto (o sin nombre)**, que se corresponde con el directorio actual
- El *paquete por defecto* suele emplearse para pequeñas aplicaciones, pruebas,... En general, a medida que nuestras aplicaciones van creciendo, trataremos de organizar sus clases en paquetes
- Más adelante, volveremos sobre el uso de paquetes en Java

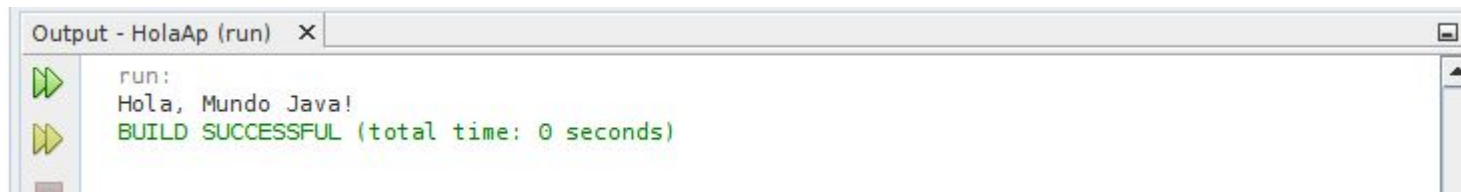
Usando un IDE (V)

- Completamos el código en el editor, añadiendo la línea:

```
System.out.println("Hola, Mundo Java!");
```

dentro del método *main*

- Finalmente, ejecutaremos nuestra aplicación desde el menú *Run > Run Project*, pulsando *F6* ó, clicando sobre el botón 
- Esta acción, compilará y ejecutará nuestro código
- En la parte inferior del IDE, en la pestaña **Output**, se mostrará la salida del mismo (o errores si los hubiera):



- Si sólo deseamos compilar para generar el *bytecode*, sin llegar a ejecutarlo, haremos *Run > Build Project*, pulsar *F11* ó, clicando 

Usando un IDE (y VI)

- Desde la pestaña *Files*, podremos observar que se ha generado un nuevo archivo, *Hola.class*, con el bytecode de la compilación del archivo *Hola.java*
- Dentro de la carpeta del proyecto, los archivos fuente se encontrarán dentro de la subcarpeta *src/nombre_paquete*
- Los archivos *.class* compilados, se encontrarán en la subcarpeta *build/classes/nombre_paquete*

