

¡HOLA  
MUNDO!

```
COBOL  
IDENTIFICATION DIVISION.  
PROGRAM-ID. HELLO-WORLD.  
PROCEDURE DIVISION.  
    DISPLAY 'Hello, world!'.  
    STOP RUN.
```

```
FORTRAN  
program hello  
    write(*,*) 'Hello, world!'  
end program hello
```

```
Lisp  
(print "Hello, world!")
```

```
Pascal  
program HelloWorld;  
  
begin  
    WriteLn('Hello, world!');  
end.
```

```
Ocaml  
print_endline "Hello, world!"
```

```
Haskell  
main = putStrLn "Hello, world!"
```

```
C  
#include <stdio.h>  
  
int main(void)  
{  
    puts("Hello, world!");  
}
```

```
Clojure  
(println "Hello, world!")
```

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

```
C++  
#include <iostream>  
  
int main()  
{  
    std::cout << "Hello, world!\n";  
}
```

```
Javascript  
console.log('Hello, world!');
```

```
SCALA  
object HelloWorld extends App {  
    println("Hello, world!")  
}
```

```
C#  
using System;  
class Program  
{  
    public static void Main(string[] args)  
    {  
        Console.WriteLine("Hello, world!");  
    }  
}
```

```
Python  
print "Hello, world!"
```

Javascript no es Java  
Está originalmente "basado"\* en objetos no "orientado"  
\* Actualmente extendido a "basado en prototipos" (o sea, sí orientado)

# Antes de ver la versión Java...

## ...una primera aproximación a la cuestión CLASE / OBJETO

**Clase** es a **tipo** como **objeto** es a **variable**

```
int var1;  
Coche miCoche;
```

**var1** es una **variable** de **tipo entero**

**miCoche** es un **objeto** de **clase Coche**



Una **clase** es un “**tipo complejo**”; una agrupación de **variables** (constantes), **objetos**, e incluso **código** que puede hacer referencias a sobre sus propios elementos u otros externos.

Un **objeto** es una cápsula (de memoria de ordenador) que tiene un “**estado**” (determinado por los valores de sus variables y el estado de sus objetos) así como un “**comportamiento**” (definido por el código que encierra).

La clase es la “definición” a partir de la cual se generan (“instancian”) los elementos –objetos– que participan en la aplicación. (esto tiene multitud de cuestiones de detalle que se irán matizando más adelante)

HolaMundo.java

```
public class GeneradorDeHolaMundo{  
    public static void main(String[] args) {  
        (java.lang) System.out.println("Hola mundo");  
    }  
}
```

Clase

Método

HolaMundo.java

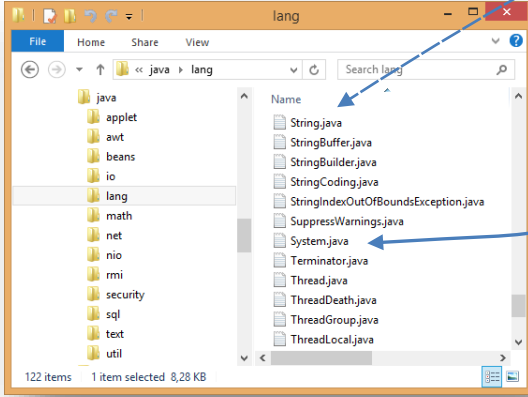
```

public class GeneradorDeHolaMundo{
    public static void main(String[] args) {
        (java.lang) System.out.println("Hola mundo");
    }
}

```

Clase

Método



System.java

```

/**
 * The <code>System</code> class contains several useful class fields
 * and methods. It cannot be instantiated.
 *
 * <p>Among the facilities provided by the <code>System</code> class
 * are standard input, standard output, and error output streams;
 * access to externally defined properties and environment
 * variables; a means of loading files and libraries; and a utility
 * method for quickly copying a portion of an array.
 *
 * @author unassigned
 * @since JDK1.0
 */
public final class System {

    /* register the natives via the static initializer.
     * VM will invoke the initializeSystemClass method to complete
     * the initialization for this class separated from clinit.
     * Note that to use properties set by the VM, see the constraints
     * described in the initializeSystemClass method.
     */
    private static native void registerNatives();
    static {
        registerNatives();
    }

    /* Don't let anyone instantiate this class */
    private System() {
    }

    /**
     * The "standard" input stream. This stream is already
     * open and ready to supply input data. Typically this stream
     * corresponds to keyboard input or another input source specified by
     * the host environment or user.
     */
    public final static InputStream in = null;

    /**
     * The "standard" output stream. This stream is already
     * open and ready to accept output data. Typically this stream
     * corresponds to display output or another output destination
     * specified by the host environment or user.
     * <p>
     * For simple stand-alone Java applications, a typical way to write
     * a line of output data is:
     * <blockquote><pre>
     *     System.out.println(data)
     * </pre></blockquote>
     * <p>
     * See the <code>println</code> methods in class <code>PrintStream</code>.
     *
     * @see java.io.PrintStream#println()
     * @see java.io.PrintStream#println(boolean)
     * @see java.io.PrintStream#println(char)
     * @see java.io.PrintStream#println(char[])
     * @see java.io.PrintStream#println(double)
     * @see java.io.PrintStream#println(float)
     * @see java.io.PrintStream#println(int)
     * @see java.io.PrintStream#println(long)
     * @see java.io.PrintStream#println(java.lang.Object)
     * @see java.io.PrintStream#println(java.lang.String)
     */
    public final static PrintStream out = null;
}

```

HolaMundo.java

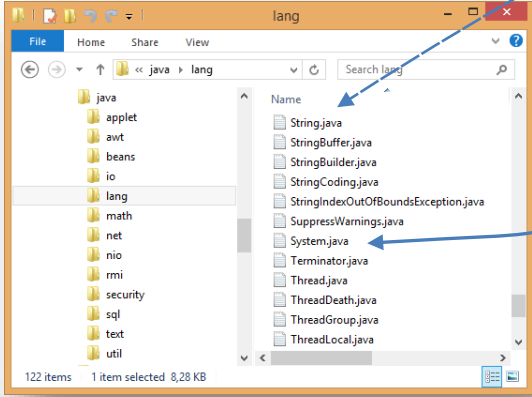
```

public class GeneradorDeHolaMundo{
    public static void main(String[] args) {
        (java.lang) System.out.println("Hola mundo");
    }
}

```

Clase

Método



System.java

```

/**
 * The <code>System</code> class contains several useful class fields
 * and methods. It cannot be instantiated.
 *
 * <p>Among the facilities provided by the <code>System</code> class
 * are standard input, standard output, and error output streams;
 * access to externally defined properties and environment
 * variables; a means of loading files and libraries; and a utility
 * method for quickly copying a portion of an array.
 *
 * @author unassigned
 * @since JDK1.0
 */
public final class System {

    /* register the natives via the static initializer.
     * VM will invoke the initializeSystemClass method to complete
     * the initialization for this class separated from clinit.
     * Note that to use properties set by the VM, see the constraints
     * described in the initializeSystemClass method.
     */
    private static native void registerNatives();
    static {
        registerNatives();
    }

    /** Don't let anyone instantiate this class */
    private System() {}

    /**
     * The "standard" input stream. This stream is already
     * open and ready to supply input data. Typically this stream
     * corresponds to keyboard input or another input source specified
     * by the host environment or user.
     */
    public final static InputStream in = null;

    /**
     * The "standard" output stream. This stream is already
     * open and ready to accept output data. Typically this stream
     * corresponds to display output or another output destination
     * specified by the host environment or user.
     *
     * <p>
     * For simple stand-alone Java applications, a typical way to write
     * a line of output data is:
     * <blockquote><pre>
     *     System.out.println(data)
     * </pre></blockquote>
     *
     * <p>
     * See the <code>println</code> methods in class <code>PrintStream</code>.
     *
     * @see java.io.PrintStream#println()
     * @see java.io.PrintStream#println(boolean)
     * @see java.io.PrintStream#println(char)
     * @see java.io.PrintStream#println(char[])
     * @see java.io.PrintStream#println(double)
     * @see java.io.PrintStream#println(float)
     * @see java.io.PrintStream#println(int)
     * @see java.io.PrintStream#println(long)
     * @see java.io.PrintStream#println(java.lang.Object)
     * @see java.io.PrintStream#println(java.lang.String)
     */
    public final static PrintStream out = null;
}

```

```

/**
 * A <code>PrintStream</code> adds functionality to another output stream,
 * namely the ability to print representations of various data values
 * conveniently. Two other features are provided as well. Unlike other output
 * streams, a <code>PrintStream</code> never throws an
 * <code>IOException</code>; instead, exceptional situations merely set an
 * internal flag that can be tested via the <code>checkError</code> method.
 * Optionally, a <code>PrintStream</code> can be created so as to flush
 * automatically; this means that the <code>flush</code> method is
 * automatically invoked after a byte array is written, one of the
 * <code>println</code> methods is invoked, or a newline character or byte
 * <code>'\n'</code> is written.
 *
 * <p> All characters printed by a <code>PrintStream</code> are converted into
 * bytes using the platform's default character encoding. The <code>@link
 * <code>PrintWriter</code> class should be used in situations that require writing
 * characters rather than bytes.
 *
 * @author Frank Yellin
 * @author Mark Reinhold
 * @since JDK1.0
 */
public class PrintStream extends FilterOutputStream
    implements Appendable, Closeable
{
    ---- código eliminado hasta el método que nos interesa ----

    /**
     * Prints a String and then terminate the line. This method behaves as
     * though it invokes <code>@link #print(String)</code> and then
     * <code>@link #println()</code>.
     *
     * @param x The <code>String</code> to be printed.
     */
    public void println(String x) {
        synchronized (this) {
            print(x);
            newline();
        }
    }
}

```

PrintStream.java

