1. In a compiled language, the target machine directly translates the program. In an interpreted language, the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code.

Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be faster and more efficient to execute than interpreted languages. While Interpreters run through a program line by line and execute each command. Here, if the author decides he wants to use a different kind of olive oil, he could scratch the old one out and add the new one.

So which one is javaScript?

Java implementations typically use a two-step compilation process. Java source code is compiled down to *bytecode* by the Java compiler. The bytecode is executed by a Java Virtual Machine (JVM). Modern JVMs use a technique called [Just-in-Time (JIT) compilation](http://en.wikipedia.org/wiki/Just-in-time_compilation) to compile the bytecode to native instructions understood by hardware CPU on the fly at runtime.

Some implementations of JVM may choose to interpret the bytecode instead of JIT compiling it to machine code, and running it directly. While this is still considered an "interpreter," It's quite different from interpreters that read and execute the high level source code (i.e. in this case, Java source code is not interpreted directly, the bytecode, output of Java compiler, is.)

It is technically possible to compile Java down to native code ahead-of-time and run the resulting binary. It is also possible to interpret the Java code directly.

To summarize, depending on the execution environment, bytecode can be:

* compiled ahead of time and executed as native code (similar to most C++ compilers)
* compiled just-in-time and executed
* interpreted
* directly executed by a supported processor (bytecode is the native instruction set of some CPUs)

Java is the first substantial language which is neither truly interpreted nor compiled; instead, a combination of the two forms is used. This method has advantages which were not present in earlier languages.

Platform-Independence

To understand the primary advantage of Java, you'll have to learn about platforms. In most programming languages, a compiler (or interpreter) generates code that can execute on a specific target machine. For example, if you compile a C++ program on a Windows machine, the executable file can be copied to any other machine but it  will only run on other Windows machines but never another machine (e.g., a Mac or a Linux machine). A platform is determined by the target machine (along with its operating system). For earlier languages, language designers needed to create a specialized version of the compiler (or interpreter) for every platform. If you wrote a program that you wanted to make available on multiple platforms, you, as the programmer, would have to do quite a bit of additional work.  You would have to create multiple versions of your source code for each platform.

Java succeeded in eliminating the platform issue for high-level programmers (such as you) because it has reorganized the compile-link-execute sequence at an underlying level of the compiler. Details are complicated but, essentially, the designers of the Java language isolated those programming issues which are dependent on the platform and developed low-level means to abstractly refer to these issues. Consequently, the Java compiler doesn't create an object file, but instead it creates a bytecode file which is, essentially, an object file for a virtual machine.  In fact, the Java compiler is often called the JVM compiler (for Java Virtual Machine).

Consequently, you can write a Java program (on any platform) and use the JVM compiler (called javac) to generate a bytecode file (bytecode files use the extension .class). This bytecode file can be used on any platform (that has installed Java). However, bytecode is not an executable file.  To execute a bytecode file, you actually need to invoke a Java interpreter (called java). Every platform has its own Java interpreter which will automatically address the platform-specific issues that can no longer be put off. When platform-specific operations are required by the bytecode, the Java interpreter links in appropriate code specific to the platform.

To summarize how Java works (to achieve platform independence), think about the compile-link-execute cycle. In earlier programming languages, the cycle is more closely defined as "compile-link then execute". In Java, the cycle is closer to "compile then link-execute".

As with interpreted languages, it is possible to get Java programs to run faster by compiling the bytecode into an executable; the disadvantage is that such executables will only work on the platform in which it is created.

So it can be summarized as JavaScript being both because a java program is first compiled into bytecode which JRE can understand. Bytecode is then interpreted by the JVM making it as interpreted language.

References

<https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/>

<https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc>

<https://stackoverflow.com/questions/1326071/is-java-a-compiled-or-an-interpreted-programming-language>

https://www.tutorialspoint.com/Why-java-is-both-compiled-and-interpreted-language