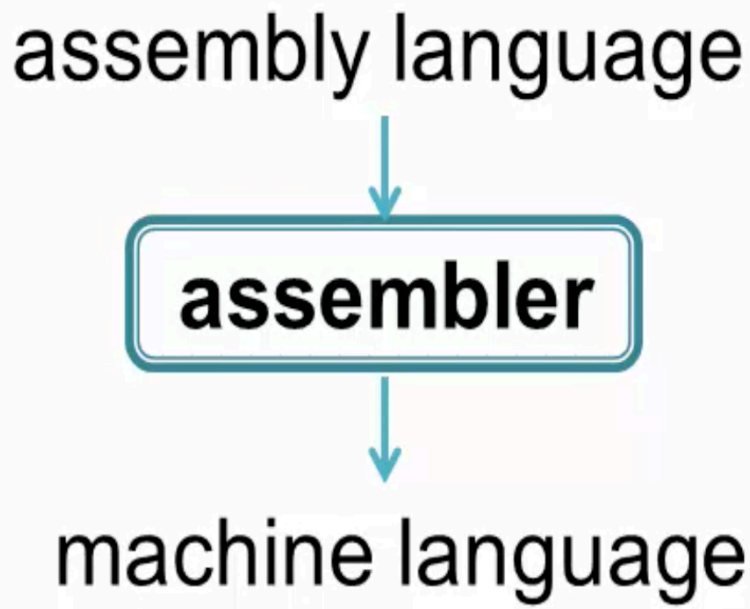
Machine Language

- Computers understand ***instructions***
- Program ➔ set of instructions
- Instruction ➔ **0s** & **1s**
- Machine language *or* machine code *or* native code

Assembly Language

- **li** $t1, 5
  add $t0, $t1, 6



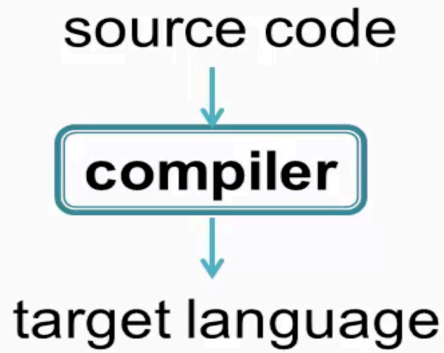
Low-level Languages

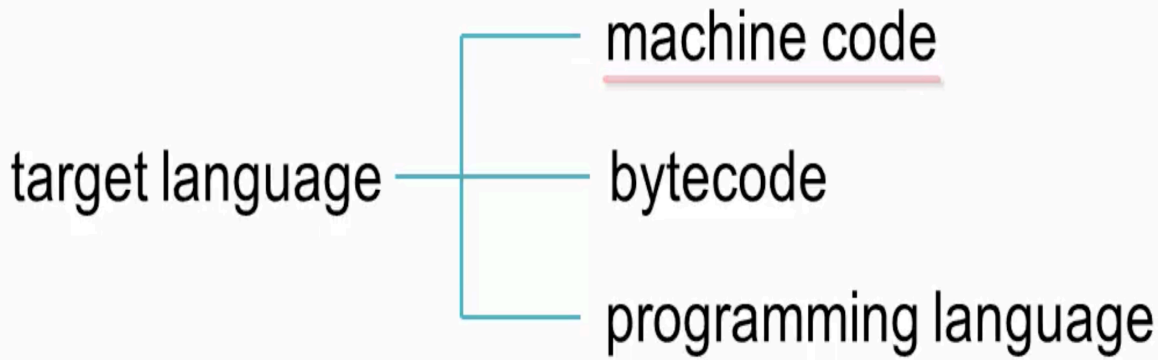
- Machine language & assembly language
- Use low-level details, e.g., memory location

High-level Languages

- FORTRAN, C, C++, Java, C#
- Use *english-like* words, *math* notations, *punctuations*
- Hide low-level details
- Source code

Compilation

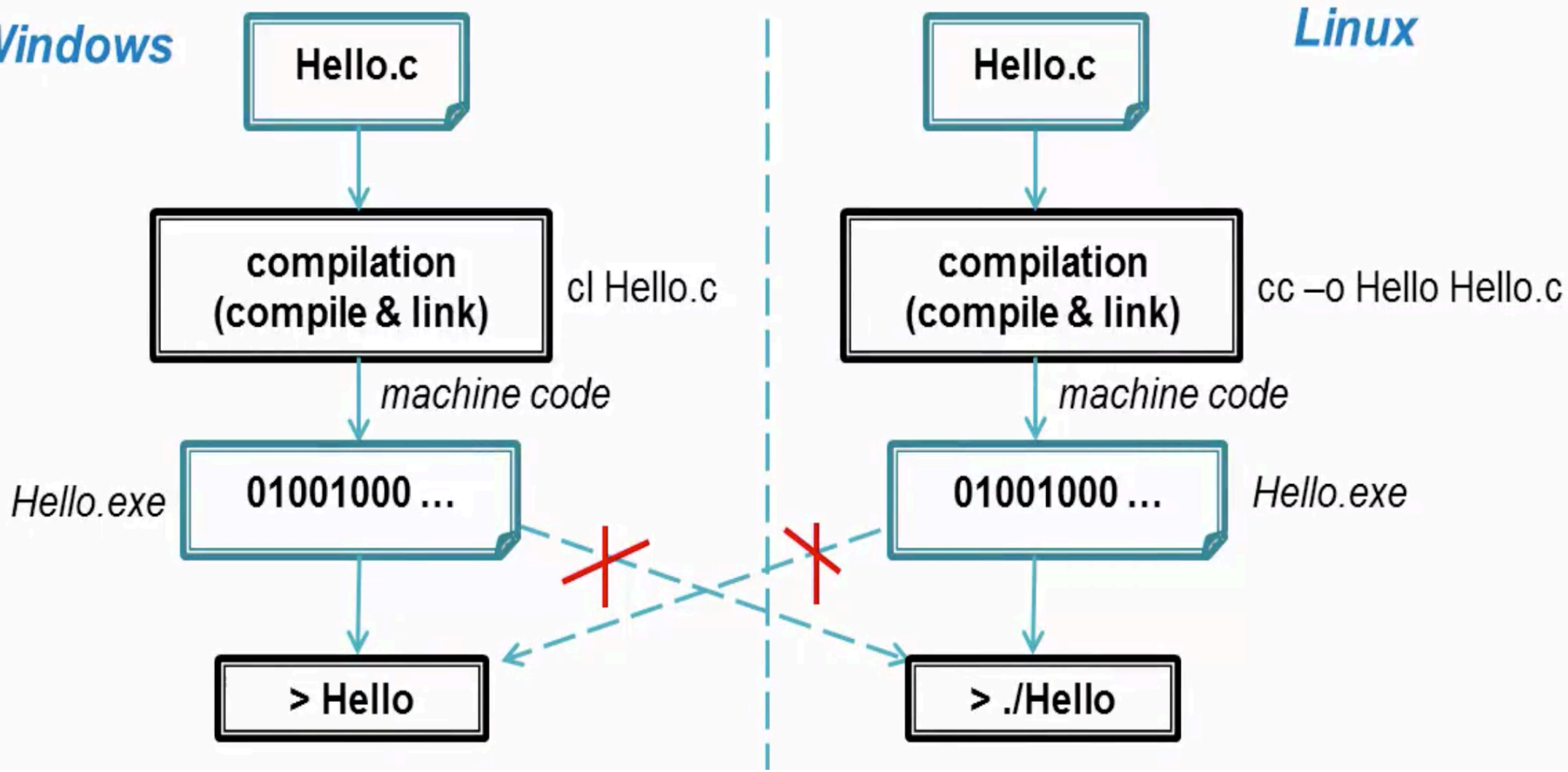




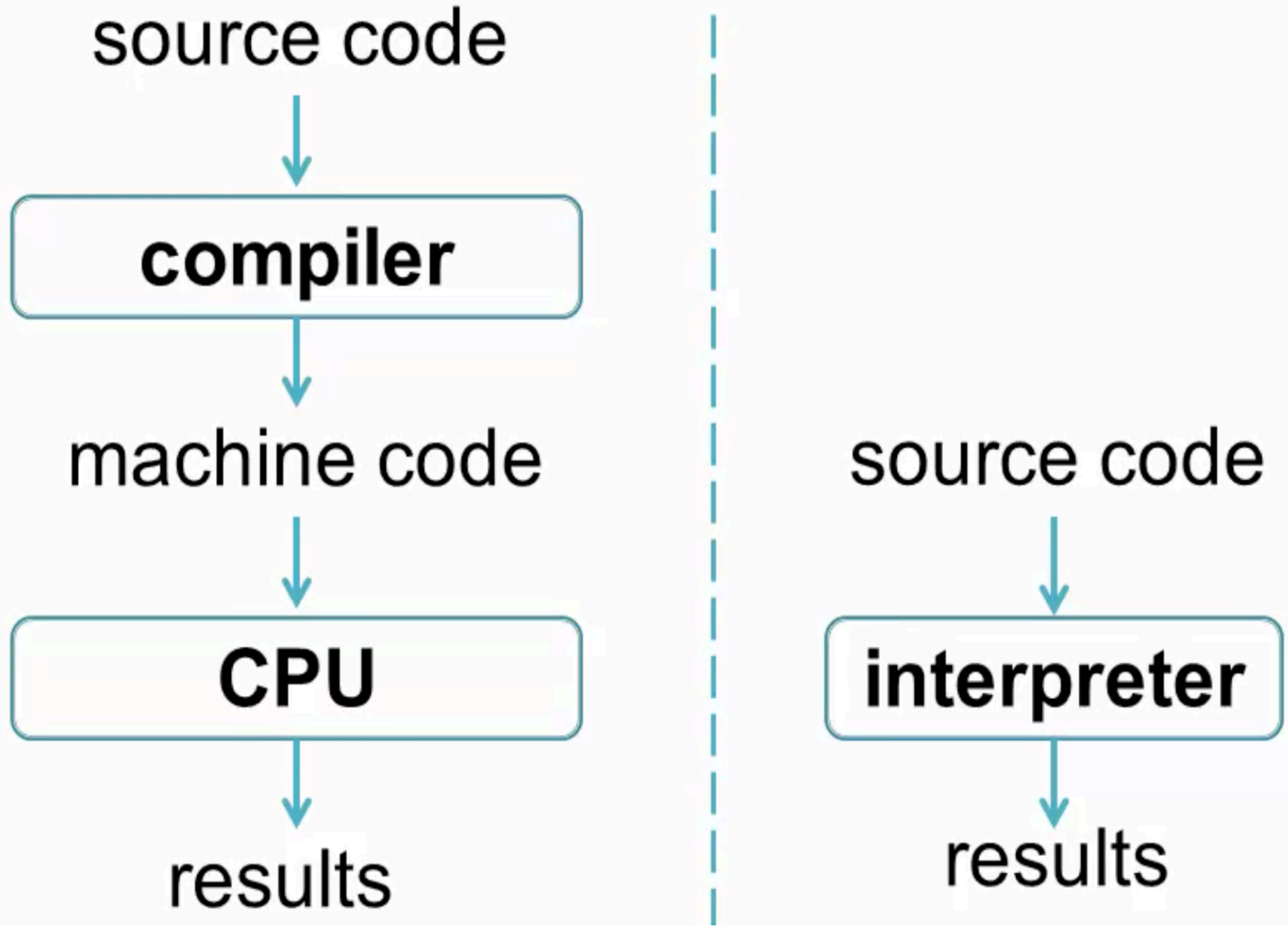
Core Compilation Operations

- Verifying **syntax** & **semantics** of source code
- Code optimizations
- Generate machine code

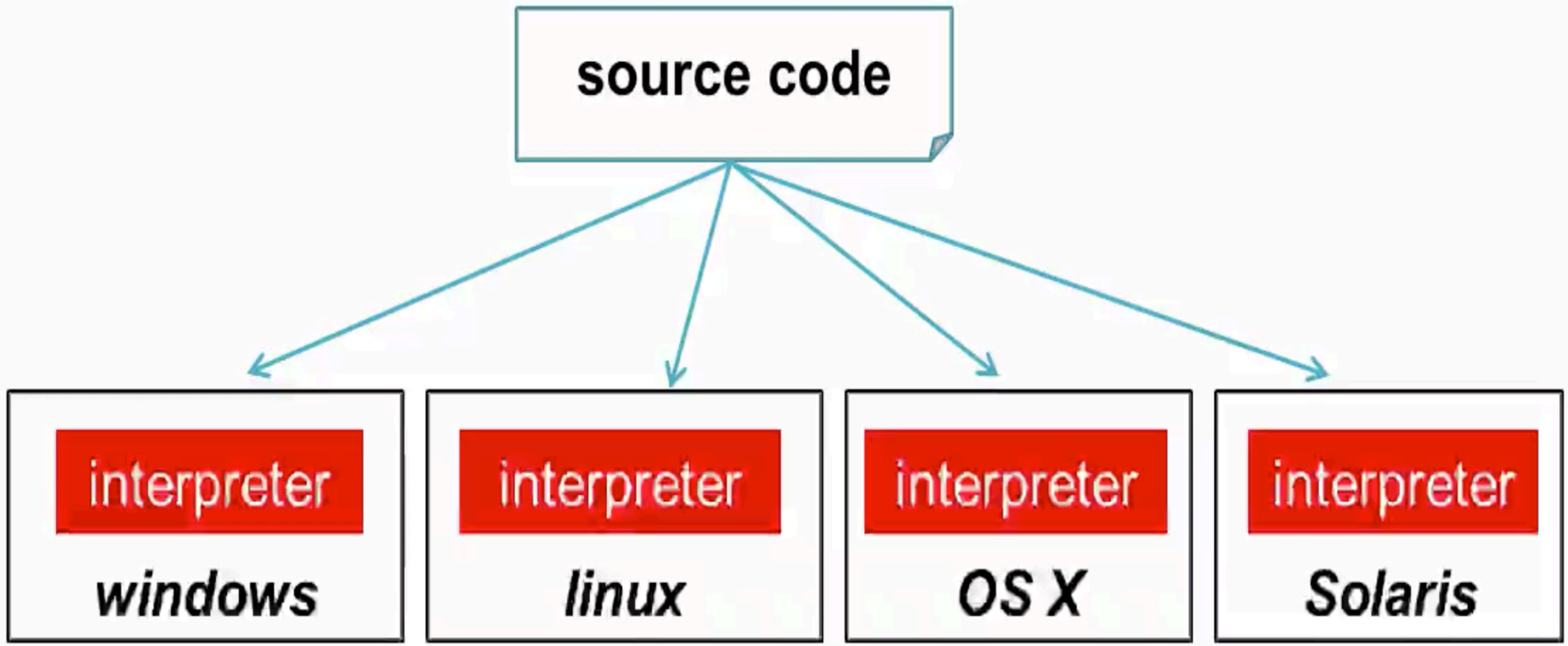
Platform Dependency



Interpretation

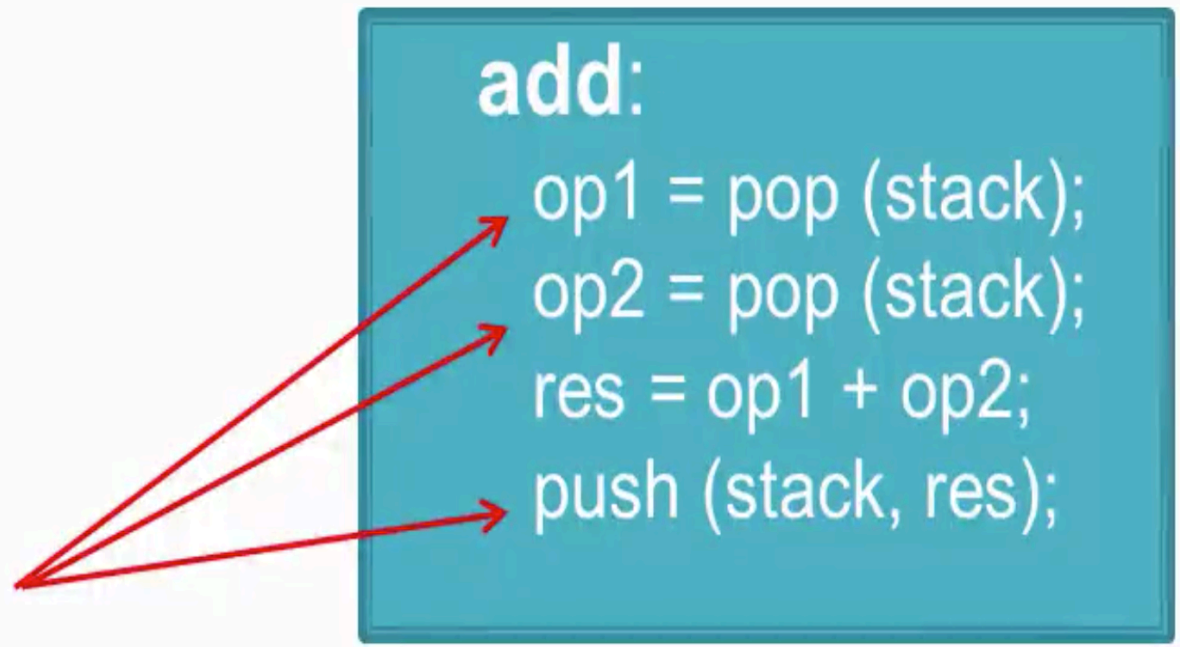


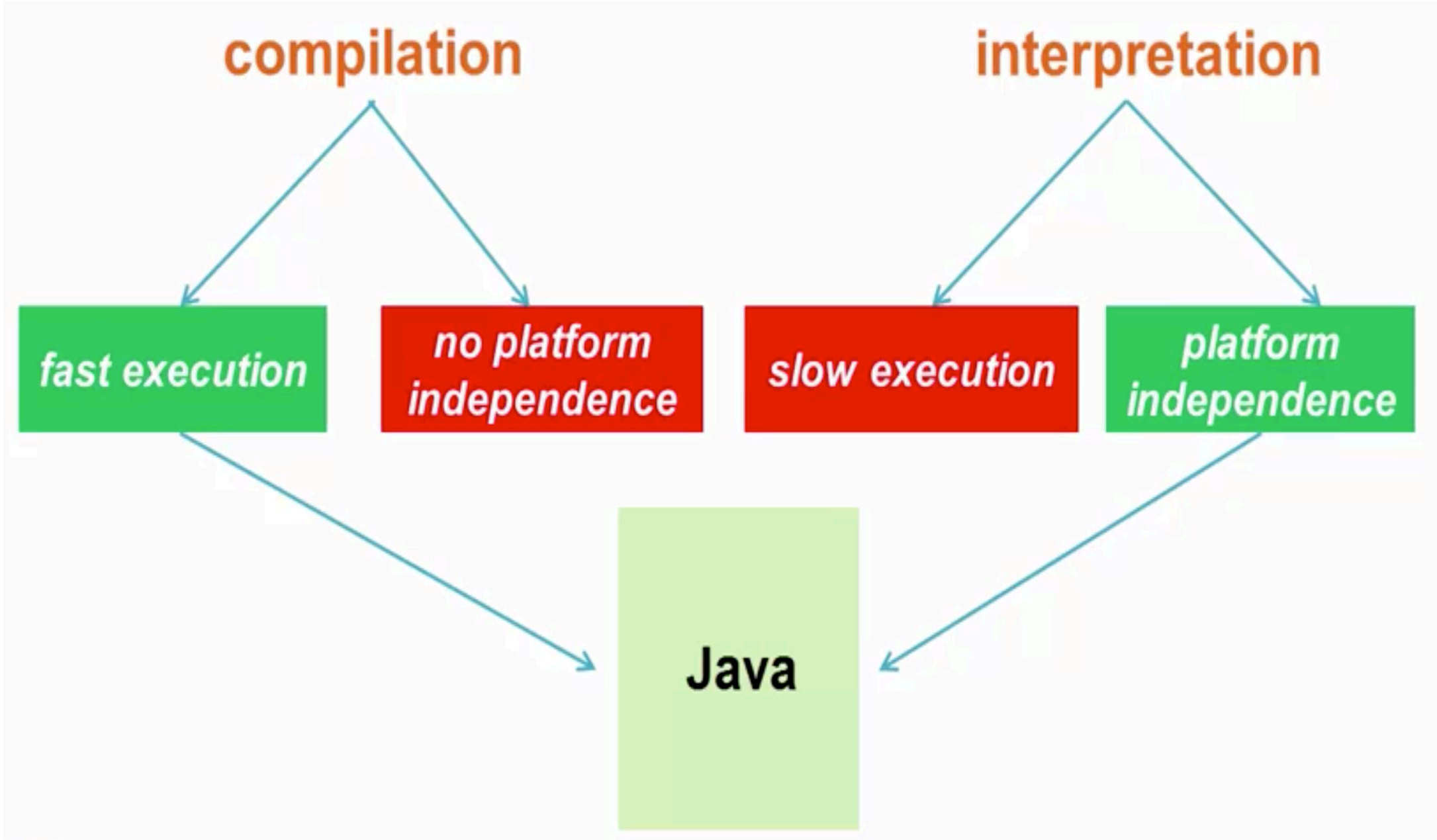
Platform Independence



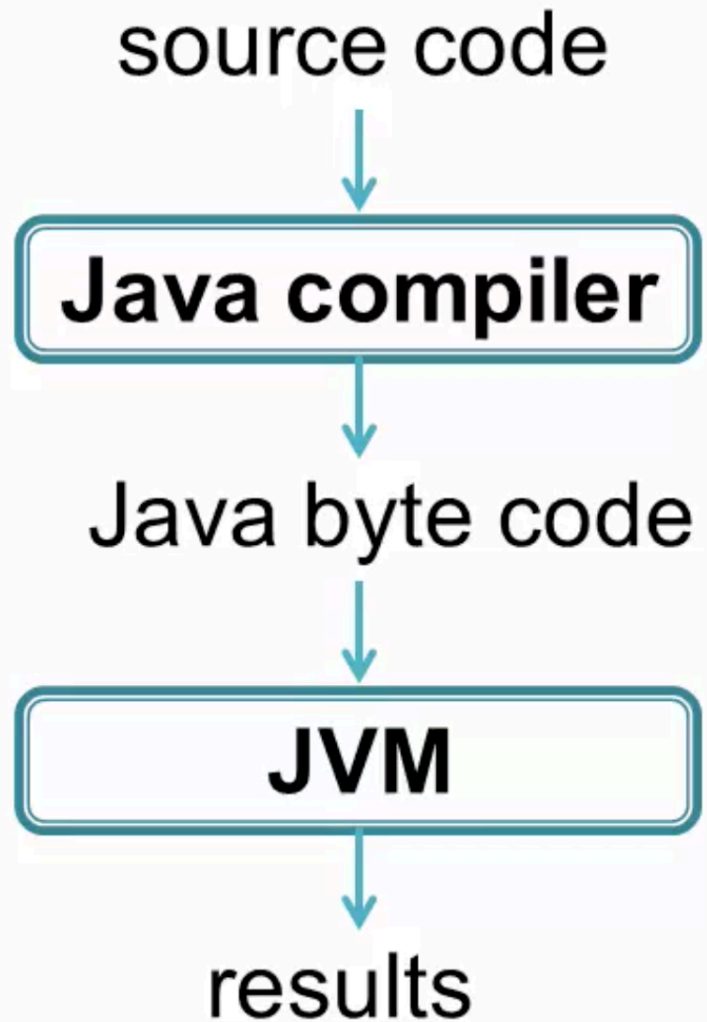
Pros & Cons

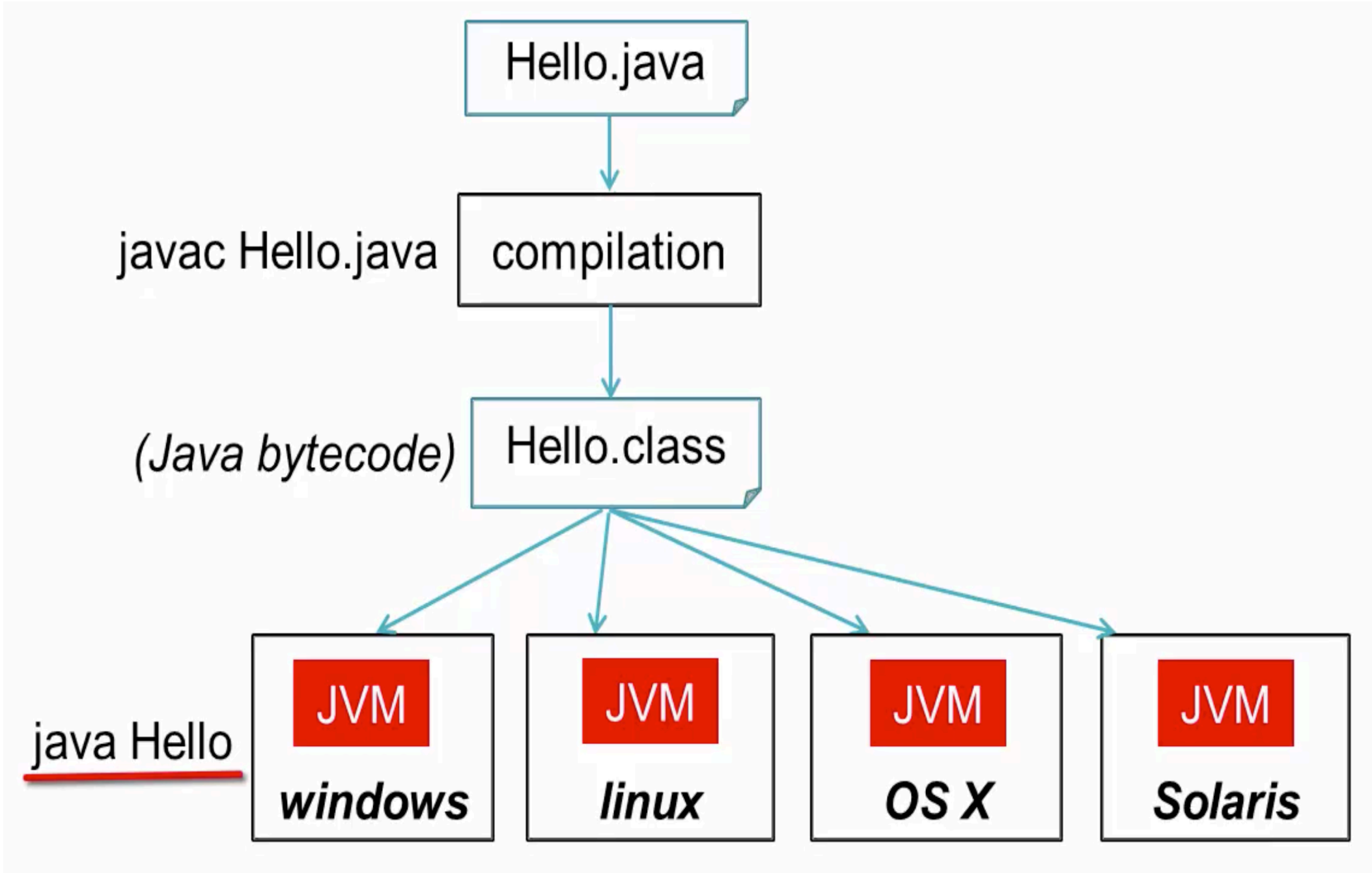
- Advantages
  - Platform independence
  - No compilation step
  - Easier to update
- Limitations
  - **Slow**
    - Costly memory access
    - Source code is reinterpreted every time
  - Interpreter is *loaded* into memory




compilation
interpretation
fast execution
no platform independence
slow execution
platform independence
Java

Bytecode Interpretation




Hello.java
javac Hello.java
compilation
(Java bytecode)
Hello.class
java Hello
JVM
windows
JVM
linux
JVM
OS X
JVM
Solaris

What about Speed?

- Bytecode interpretation is much faster
  - Java bytecode is *compact*, *compiled*, and *optimized*
- Just-in-time (JIT) compilation

Bytecode compactness → quick transfer across networks

Java Virtual Machine

Cornerstone of Java platform

Abstract Computing Machine

- ✓ Instruction set → Java bytecode
- ✓ Manipulates memory at runtime

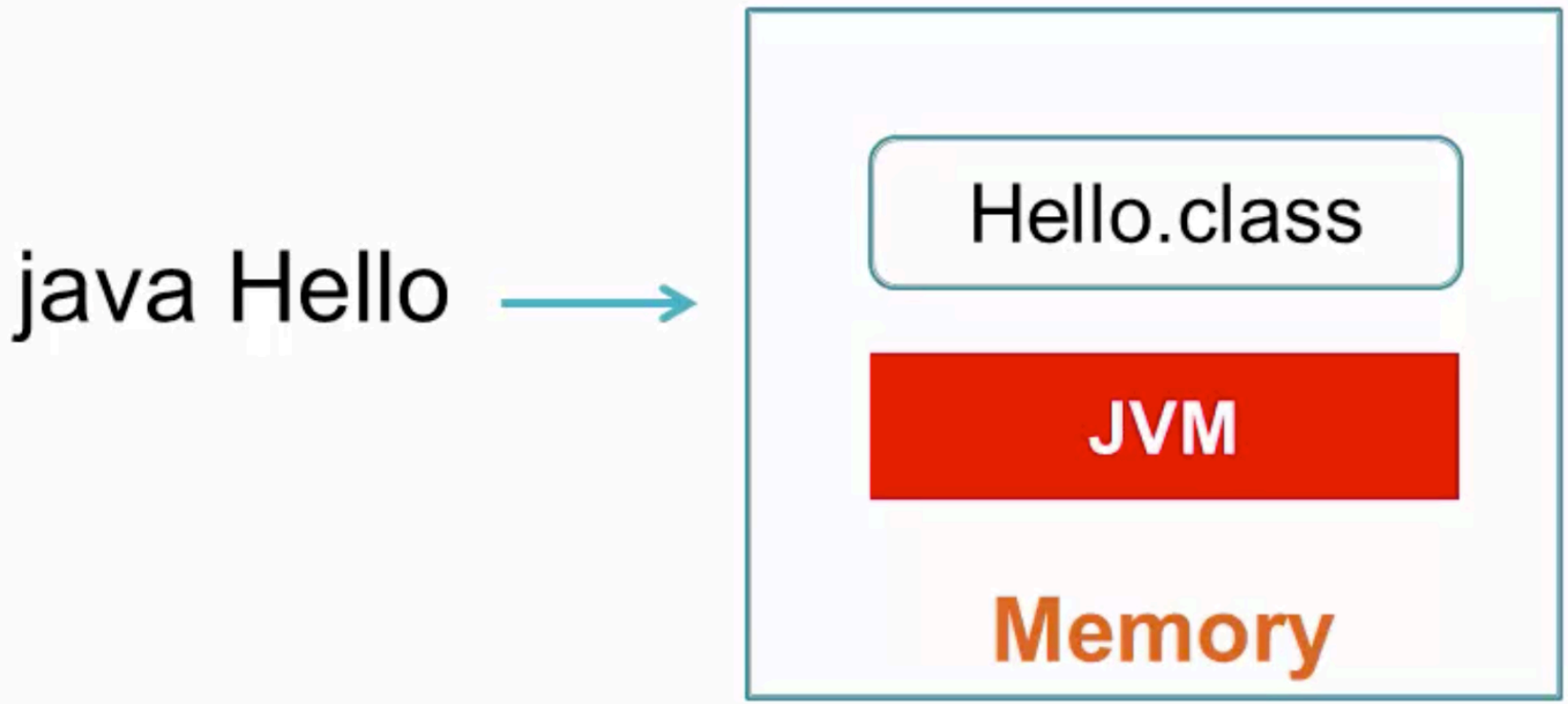
Core Responsibilities

- Loading & interpreting bytecode
- Security
- Automatic memory management

Specification & Implementation

- Abstract JVM specification
- Concrete implementation
  - Oracle's HotSpot JVM
  - IBM's JVM
- Runtime instance

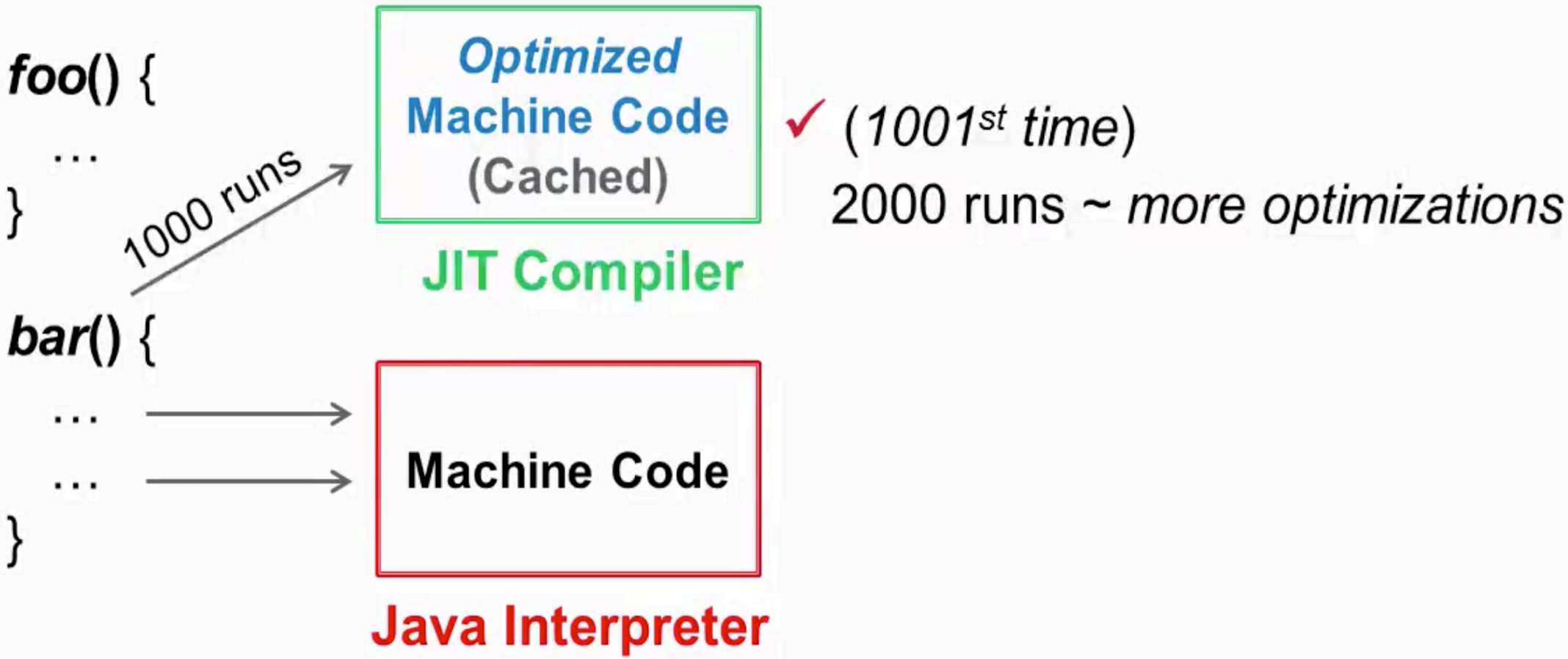
Runtime Instance



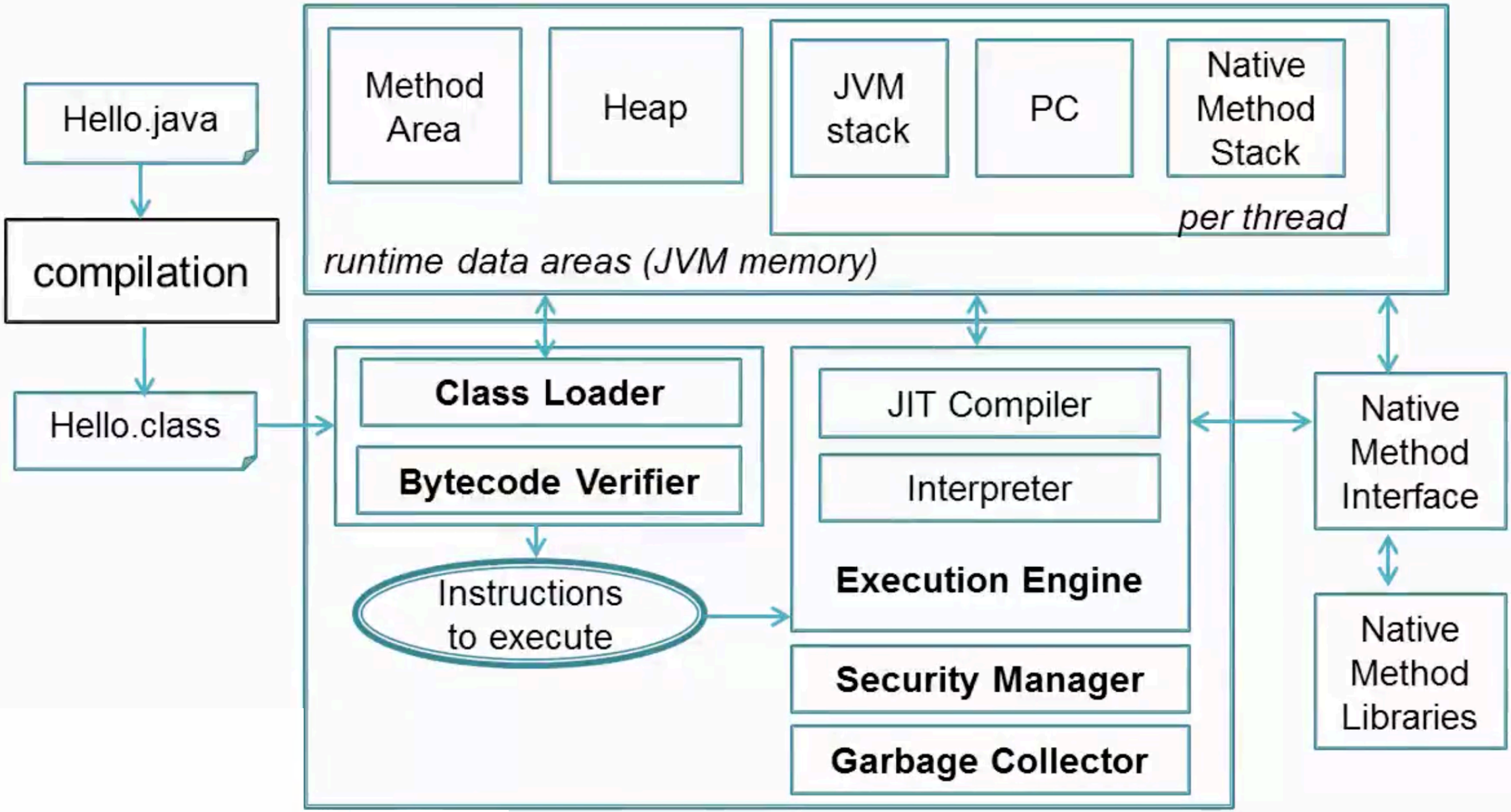
Performance

- Bytecode interpretation is much faster
  - Java bytecode is *compact*, *compiled*, and *optimized*
- Just-in-time (**JIT**) compilation
  - Identify frequently executed bytecode ~ "hot spots"
  - JIT compiler converts "hot spots" to machine code
  - *Cache* machine code
  - Cached machine code ➔ faster execution
  - Also called **dynamic compilation**

JIT Compilation Example



JVM Architecture



Below are some useful references pertaining to JIT compiler. They mainly explain whatever is described in the lecture with regards to JIT compiler. So, if you are interested you may check them out too.

1. http://stackoverflow.com/questions/3718024/jit-vs-interpreters ~ KGhatak's answer is excellent in that it explains the difference between Java's normal interpretation & JIT's compilation. Both generatemachine code, but JIT's version is optimized. Moreover, a method of block of code that is JIT compiledwill not be reinterpreted and will directly use the cached JIT generated machine code

2. Check out Paul Pacheco's answer https://www.quora.com/If-a-JIT-compiler-compiles-the-bytecode-into-machinecode-what-is-the-interpreter-doing-Does-it-simply-run-the-machine-code-Do-JITand-the-interpreter-work-in-parallel

3. Checkout Vicky Singh's answer https://www.quora.com/In-Java-what-exactly-will-the-JVM-interpreter-and-theJIT-compiler-do-with-the-bytecode

4. http://stackoverflow.com/questions/2377273/how-does-an-interpretercompiler-work?lq=1

Java Software Family

- Java Standard Edition (Java SE)
  - Standalone applications for desktops & servers, e.g., *inventory management system* in a hardware store
- Java Enterprise Edition (Java EE)
  - *Enterprise* applications for servers, e.g., *e-commerce Websites*
  - Includes Java SE
- Java Micro Edition (Java ME)
  - Applications for *resource-constrained* devices

Java SE

- Java Runtime Environment (**JRE**)
  - Only *run* Java programs

- Java Development Kit (**JDK**)
  - *Develop* and *run* Java programs

Java Versions

- **jdk 1.0**: 250 classes, slow, Applets were big ~ Jan '96
- jdk 1.1: 500 classes, little faster ~ 1997
- **J2SE 1.2**: Over 1500 classes, much faster ~ 1998
- J2SE 1.3 ~ 2000
- J2SE 1.4 ~ 2002
- **J2SE 5.0**: 2500 classes, lot of additions like *generics*, *enums*, *annotations* ~ 2004

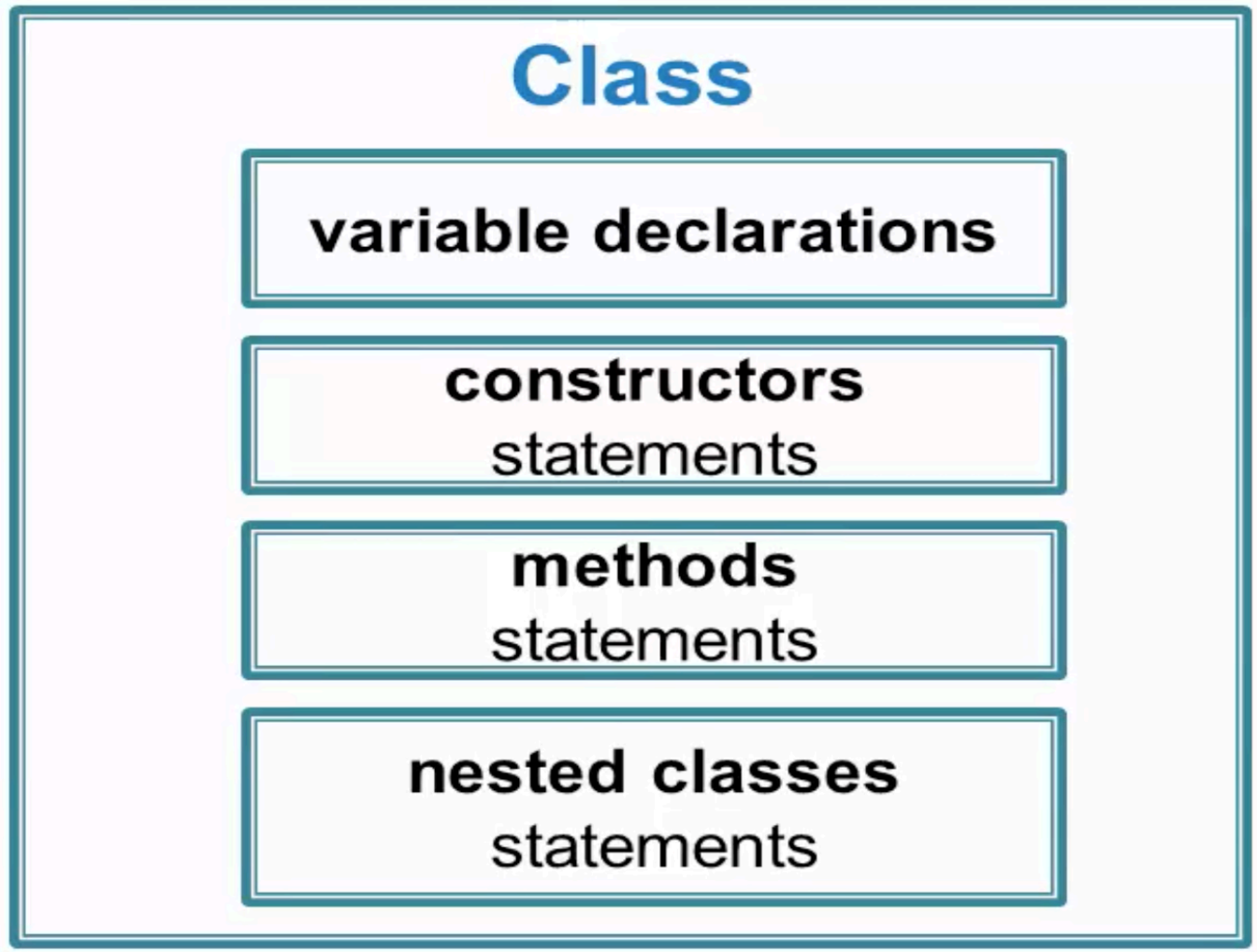
Java Versions

- Java SE 6: new bytecode verification process ~ 2006
- **Java SE 7**: better support for other languages ~ 2011
- *Java SE 8*: lambda expressions, streams, improved date and time libraries, etc. ~ 2014
- Java SE 9 ~ 2017
- OpenJDK 6 – OpenJDK 9

Setting-up Java

- ✓ Install Java
- ✓ Make **command prompt** aware of Java executables
  - Set **Path** & **JAVA_HOME** environment variables

Java Program: Structure

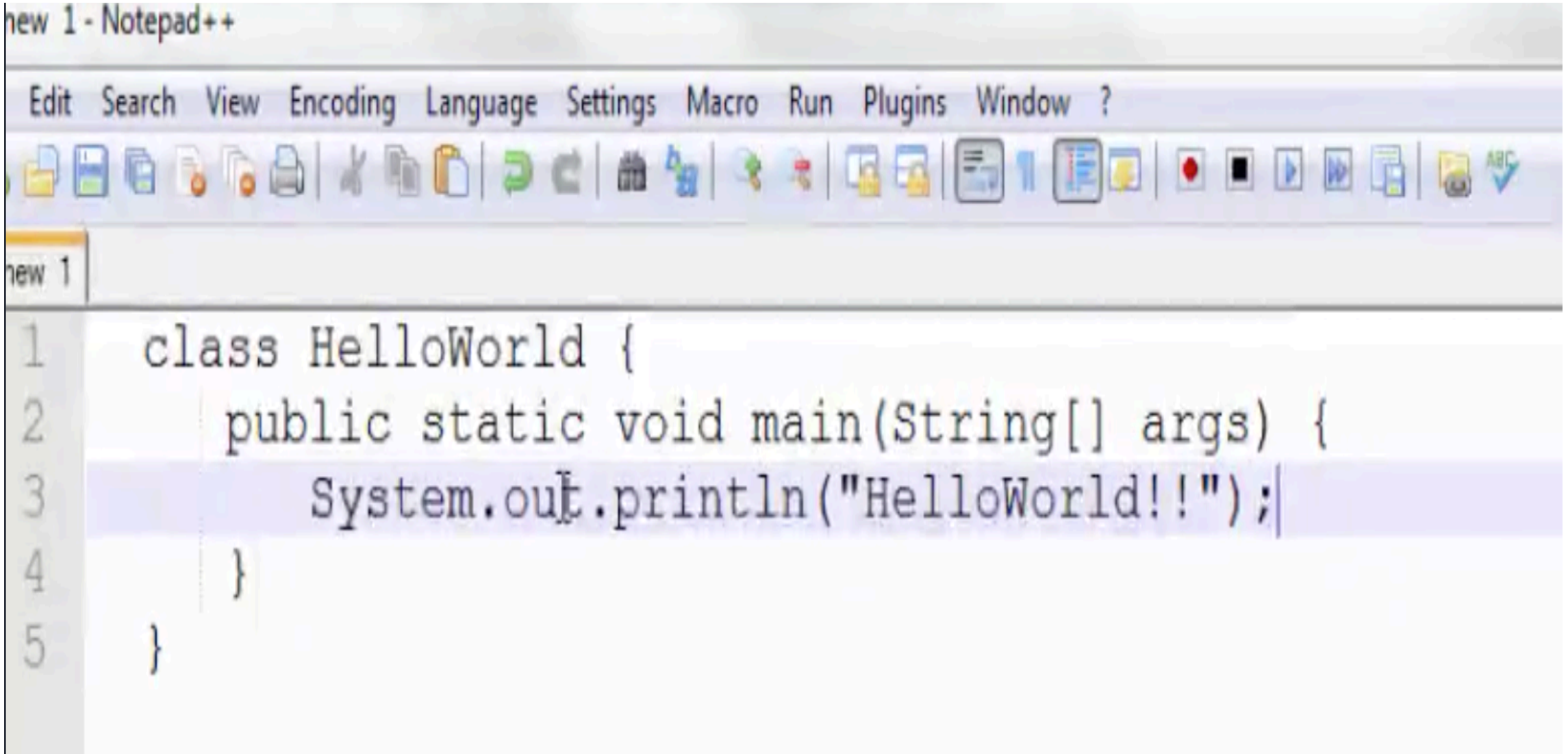


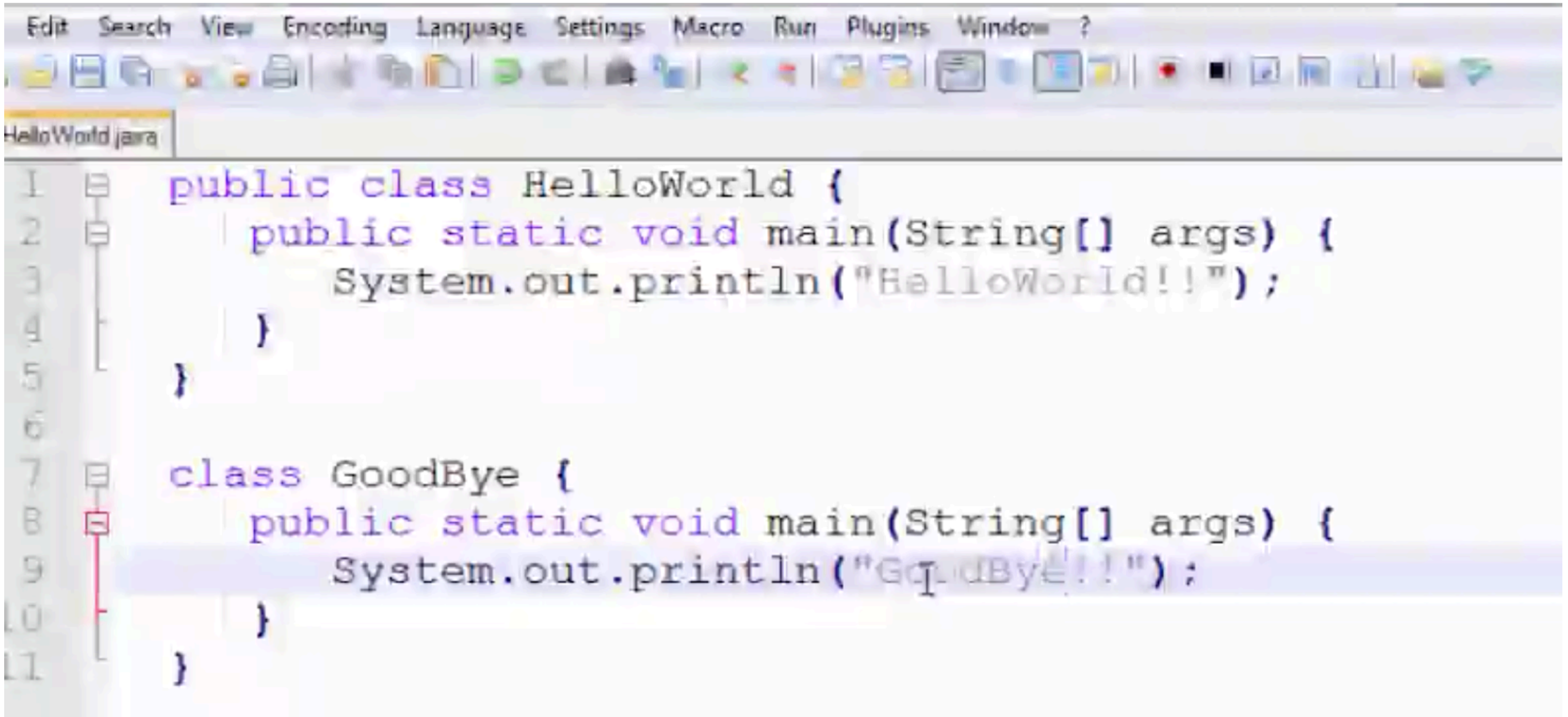
You can create the directory structure in root directory. Below are specific instructions on Windows and Linux.
Windows: - Open command prompt and go into C:\ drive.
Now, when you open command prompt, if the default directory path is not C:\ and is something else like say C:\dir1\dir2, then type cd.. to move into dir1 and one more cd.. to move into C:\ - In C:\ drive, type below instructions one after another.
mkdir is for creating a directory in the current directory while cd helps you to move into that directory:

```
mkdir javacourse
cd javacourse
mkdir src
cd src
mkdir com
cd com
mkdir training
cd training
mkdir basics
cd basics
```

Linux: Just run below command.

```
mkdir -p javacourse/src/com/training/basics && cd training/src/com/training/basics
```


new 1 - Notepad++
Edit Search View Encoding Language Settings Macro Run Plugins Window ?
new 1
1 class HelloWorld {
2 public static void main(String[] args) {
3 System.out.println("HelloWorld!!");
4 }
5 }


Edit Search View Encoding Language Settings Macro Run Plugins Window ?
HelloWorld.java
1 public class HelloWorld {
2 public static void main(String[] args) {
3 System.out.println("HelloWorld!!");
4 }
5 }
6
7 class GoodBye {
8 public static void main(String[] args) {
9 System.out.println("GoodBye!!");
10 }
11 }

main() method

- Program starts with main()
- java HelloWorld
  - JVM loads bytecodes of HelloWorld.class into memory
  - Invokes **main()**
- Must be declared as *public*, *static*, and *void*
- From main() we invoke other code
- Program ends with main()

Java Principles

- **Robustness**
  - Bytecodes are verified by *bytecode verifier*
- **High performance**
  - Bytecode's compact form + JIT compilation
- **Familiar & Simple**
  - Similar to C & C++, garbage collection
- **Object-oriented** & **Multi-threaded**