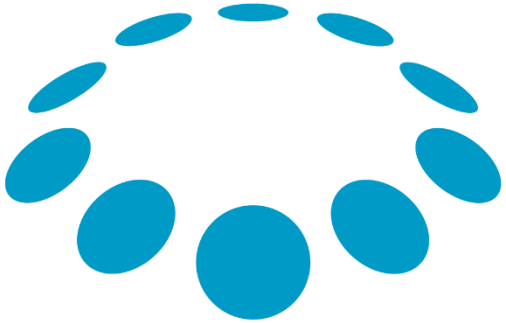
Introduction to Java

Dr. Eliahu Khalastchi

2017



המסלול האקדמי
המכללה למינהל

Short History

- First developed as an internal project in SUN in the early 90's
- The goal: use the same code in different platforms
- Almost pure **object oriented** language
- Similar syntax to C++ (why?)
- A **W**orld **W**ide **W**eb language
- Version 8

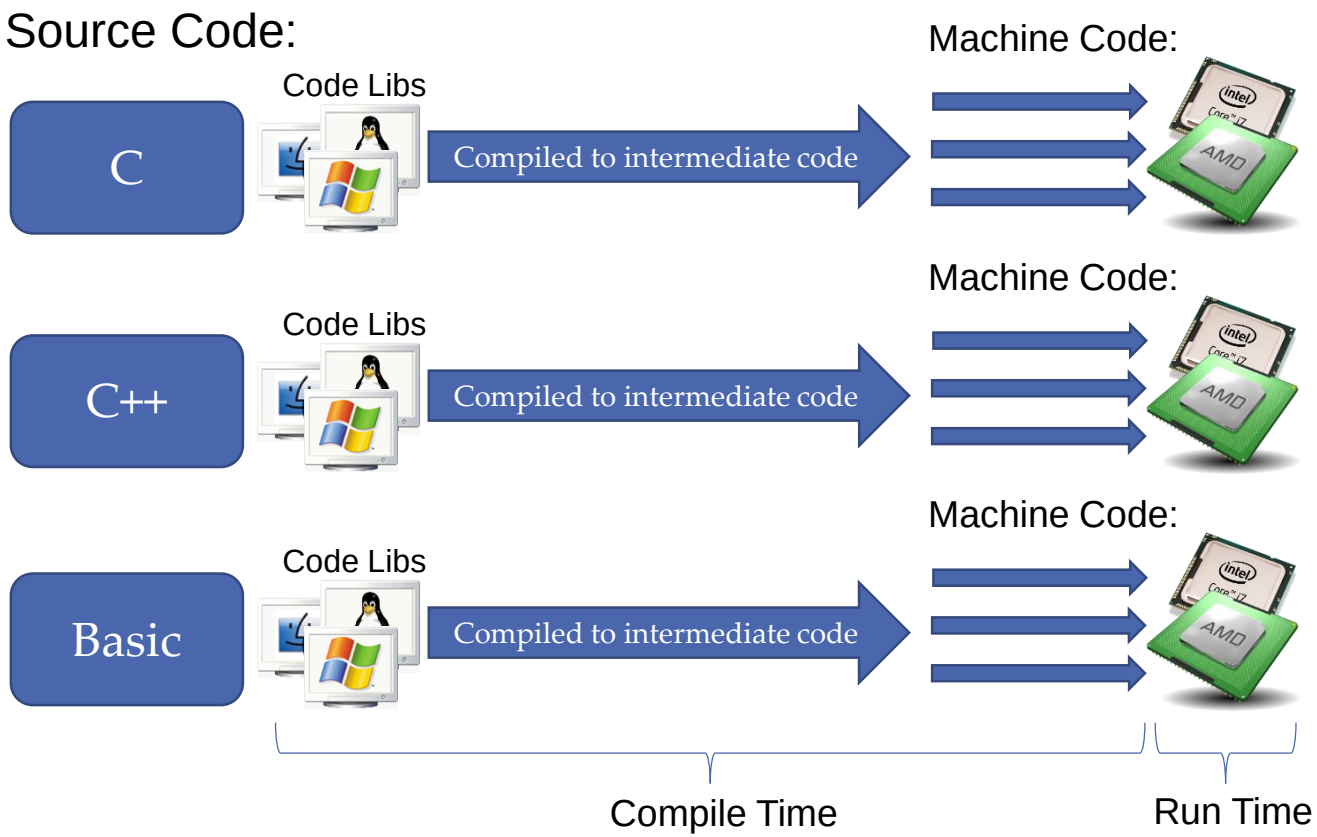
Java editions

- Standard Edition (SE)
  - For general purpose use on desktop PCs, servers and similar devices
- Enterprise Edition (EE)
  - Java SE plus various APIs useful for client–server enterprise applications (mainly server side)
- Micro Edition (ME)
  - A portion of the SE libraries, useful for cellular apps.

Why Java?

- Widely used in the industry
- Write software on one platform and run it on virtually any other platform
- Create programs to run within a Web browser and Web services
- Develop server-side applications
- Write powerful and efficient applications for mobile phones (android)

Traditional Architecture – using compilers



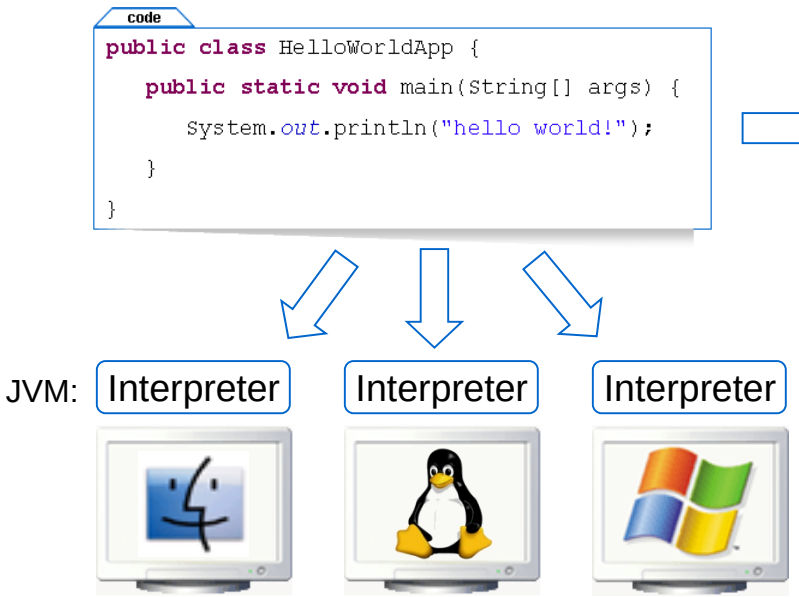
- Really hard to make a program portable
  - Recompile with new system / user libraries for a specific compiler
  - Harder to reuse code written in other languages
- Must implement your own infrastructure
  - Memory management
  - Threading
- Or be highly dependent on the operating system services

What is a JVM?

- JVM – java virtual machine
- The basic idea:
  - Instead of compiling your code to native code
  - The code is "compiled" to the **JVM's native code**
  - It can run on any machine with a JVM
- Mainly used for the Java language

- Today new languages are JVM based
  (Scala, Groovy, Jython, JRuby)

JVM architecture



- The JVM has an interpreter
- But it does not interpret High Level code

javac:

Compiler → ByteCode

- **This is way too slow…**

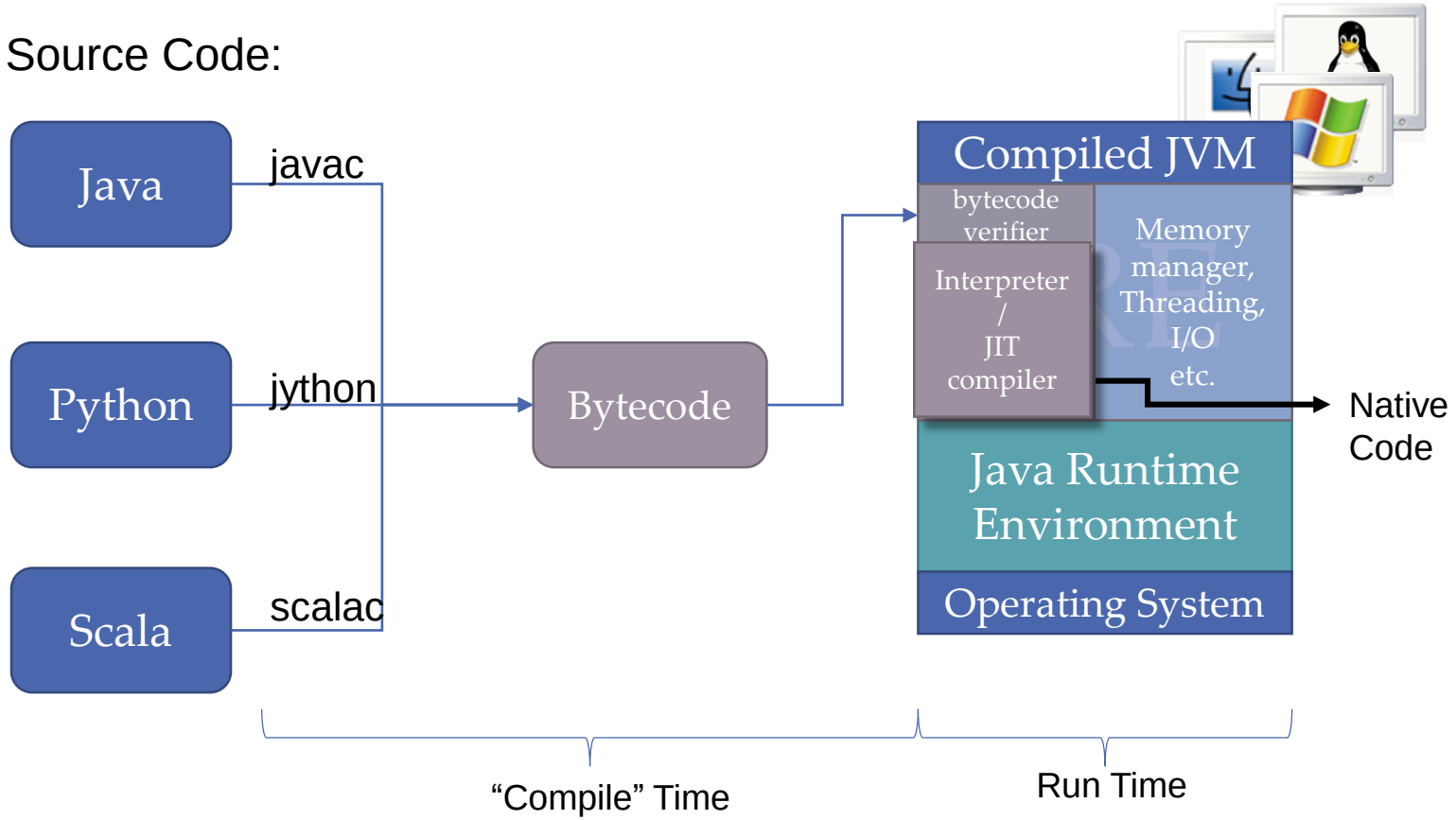
```
.class public HelloWorldApp
.super java/lang/Object

.method public static main : ([Ljava/lang/String;)V
   .limit stack 10
   .limit locals 10

   getstatic java/lang/System out Ljava/io/PrintStream;
   ldc "hello world!"
   invokevirtual java/io/PrintStream println (Ljava/lang/Object;)V
   return
.end method
```

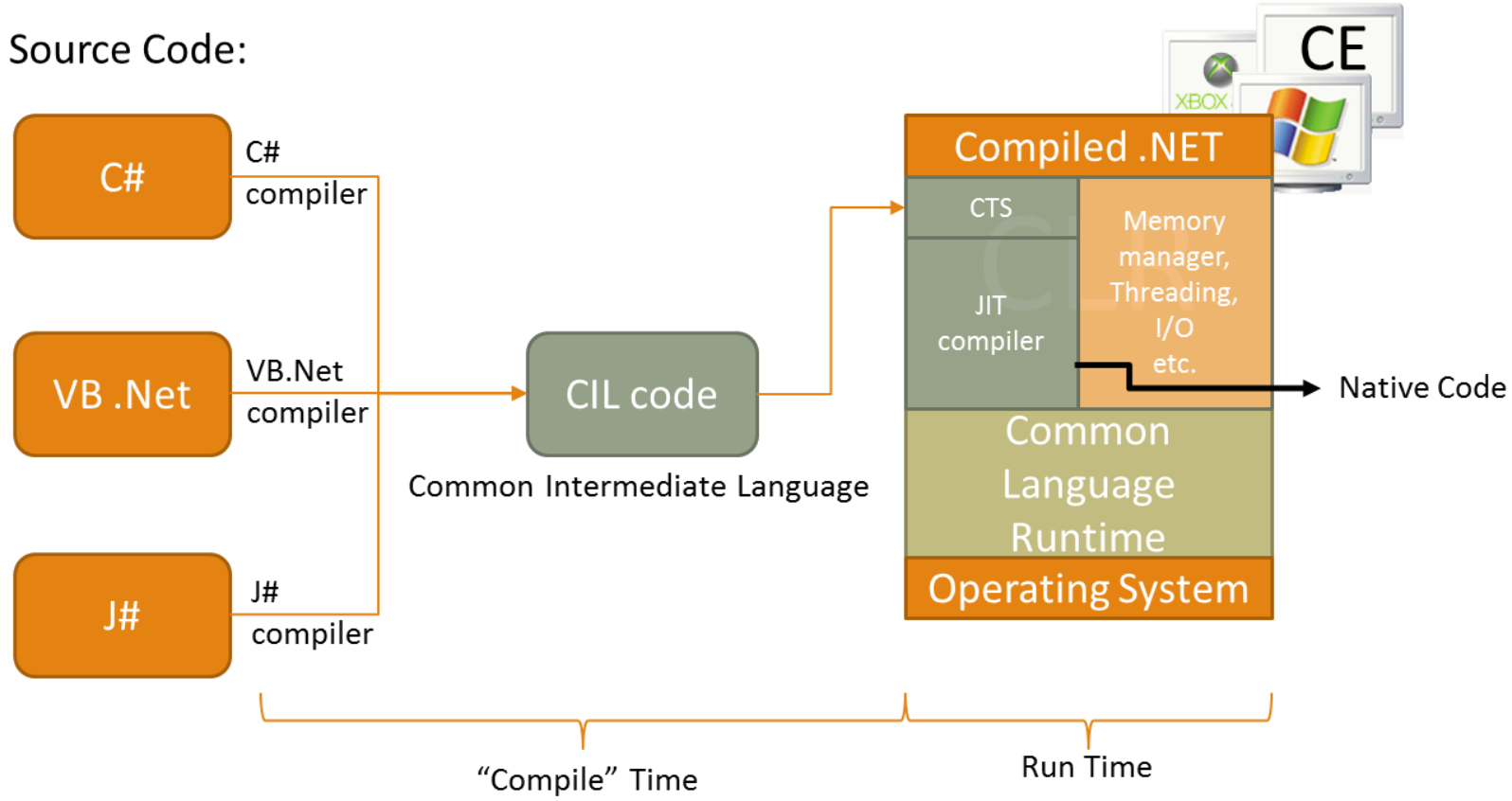
- Therefore a "compilation" phase was added
- high-level code → byte code
- byte code – a compressed low-level code
- Improved performance

The JVM Architecture – multiplatform



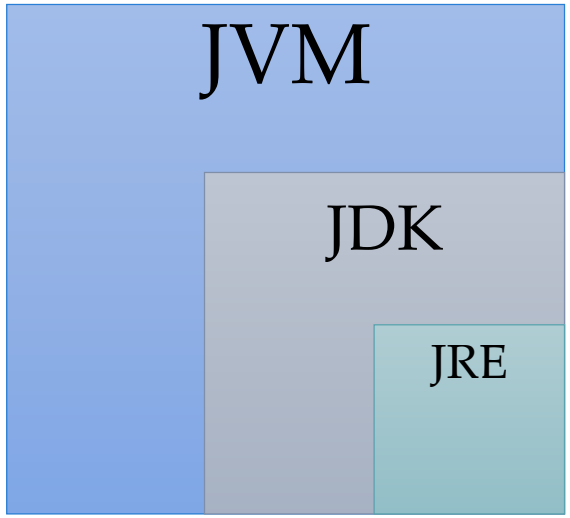
- A multiplatform architecture
  - No need to change source code to run on a different system as long there is a JVM on it
- Can load classes at runtime
  - Regardless of their source code
- A managed environment
  - E.g. a garbage collection

The .NET Architecture



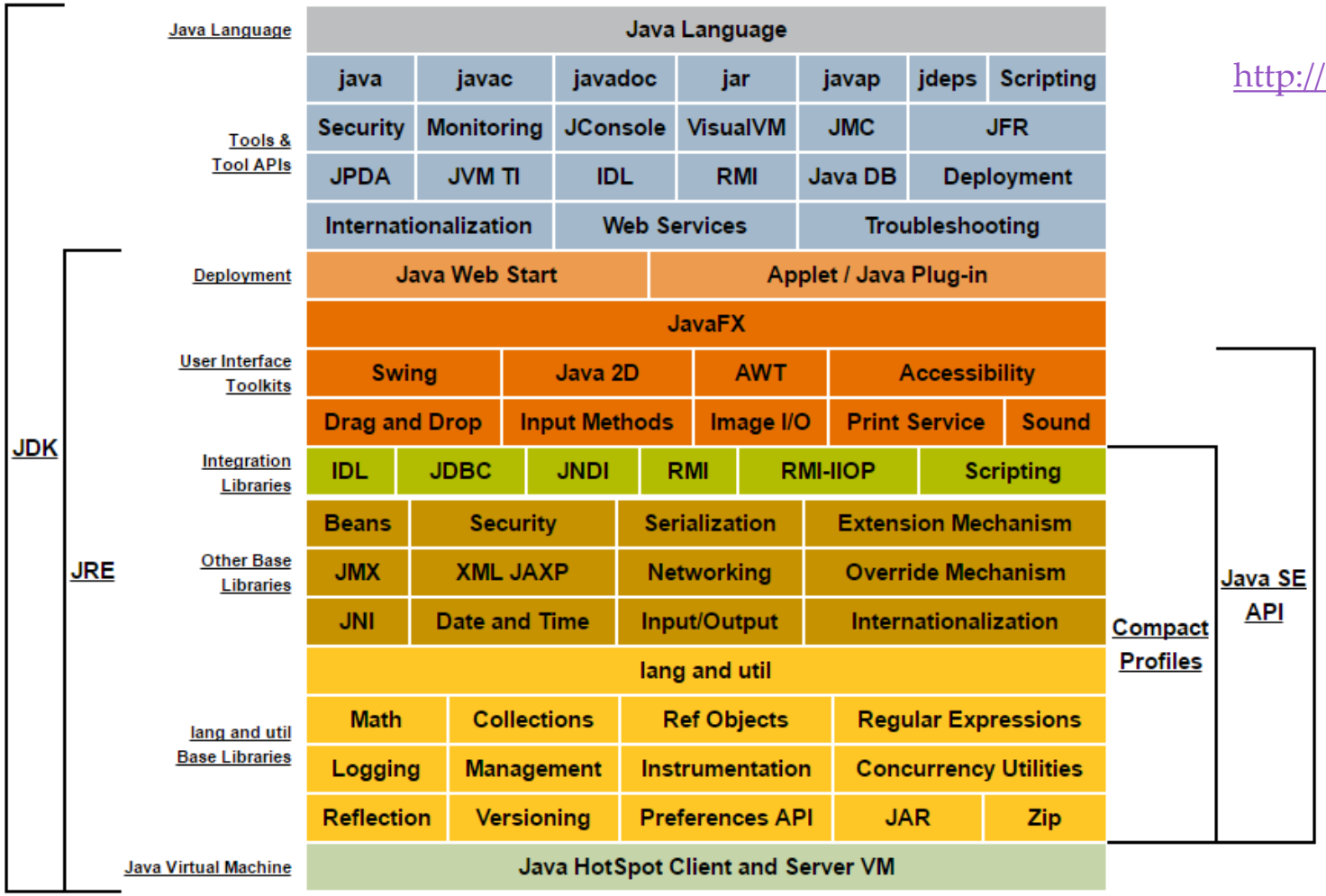
JVM architecture

- JRE – Java's Runtime Environment
  - This is the JVM (implemented for every machine)
  - Contains the java interpreter (java command)
  - The very basic libraries for java's syntax
- JDK – Java's Development kit
  - Contains all the edition's libraries
  - Contains the java compiler (javac command)



The following conceptual diagram illustrates the components of Oracle's Java SE products:

Description of Java Conceptual Diagram



http://docs.oracle.com/javase/8/docs/

Java Files

- .java – a java source code (usually contains one class)
- .class – a java byte code (“compiled” source)
- .jar – Java Archive - used for:
  - A Java library that can be imported / exported
  - A runnable file

So how does it work?

- We wrote the source code in "**HelloWorld.java**"
- We compiled the code using: "**javac HeloWorld.java**"
- The result is "**HelloWorld.class**"
  - This is the byte code
- We run the program using: "**java HelloWorld**" (without an extension)
- The JVM loads the .class file and searches for an entry point

So, how does it work?

- Each Java project has at least one entry point
- An entry point is a static method with the following signature:

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

- Static members (methods and fields)
  - are accessible from the class itself
  - And not from the class's instance = object
- Why does the main needs to be static?

HelloWorldApp.Java

code

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

C:\> javac HelloWorldApp.java

javac

HelloWorldApp.class

```
.class public HelloWorldApp
.super java/lang/Object

.method public static main : ([Ljava/lang/String;)V
   .limit stack 10
   .limit locals 10

   getstatic java/lang/System out Ljava/io/PrintStream;
   ldc “hello world!"
   invokevirtual java/io/PrintStream println (Ljava/lang/Object;)V
   return
.end method
```

This file represents a class

C:\> java HelloWorldApp

We run the

JVM

The argument is the class file as **an input** to the JVM

Loads the class into memory

- All we have is a class.
- There are no objects yet.
- From a class we can only run static methods!
  Thus the main has to be static

The JVM finds the main method according to its signature, and starts running it.

The main creates objects. These object interact, and our program runs.

So, how does it work?

- Now, the byte code (in the .class file) is being interpreted by the JVM
  - Each line is interpreted and executed

- Consider the following code

```
for(int i=0; i<n; i++){
 action1();
 action2();
 action3();
}
```

- What is the problem? Can be optimized?

JIT – Just In Time!

- Introducing JIT!
- Just In Time – compiler
  - Runtime compilation! (yes, there is such a thing)

- Until now, we knew pre-runtime compilation
- JIT offers in runtime to compile parts of the code into native (machine) code
  - only when it is worth while to do so

- So why not compile the entire code before runtime?
  - What are the benefits of runtime compilation?

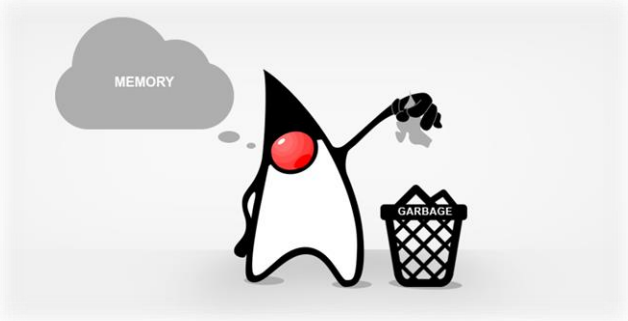
JIT – Just In Time!

- In runtime we have more information
- For example, we know what *n* is (user's input)

- JIT can decide that it is better to compile the loop's code and "inline" it n times
- Also avoids asking i<n each iteration…
- If n is to small JIT can decide it is faster to just interpret it.

GC – Garbage Collector

- Java introduced the garbage collector mechanism
- The programmer is exempt from having to free allocated memory (for obvious reasons)
- The GC automatically frees objects that are not referenced by any pointer
- Can be invoked by System.gc();
  - Will free all unreferenced objects



GC – Garbage Collector

- Each class can override the inherited ***finalize()*** method
- When the GC frees an object it calls finalize()
- Used for proper closer of an object before being freed

- Remember, the programmer does not have control when this method is being called by the GC
  - Unless the programmer invoke it explicitly