

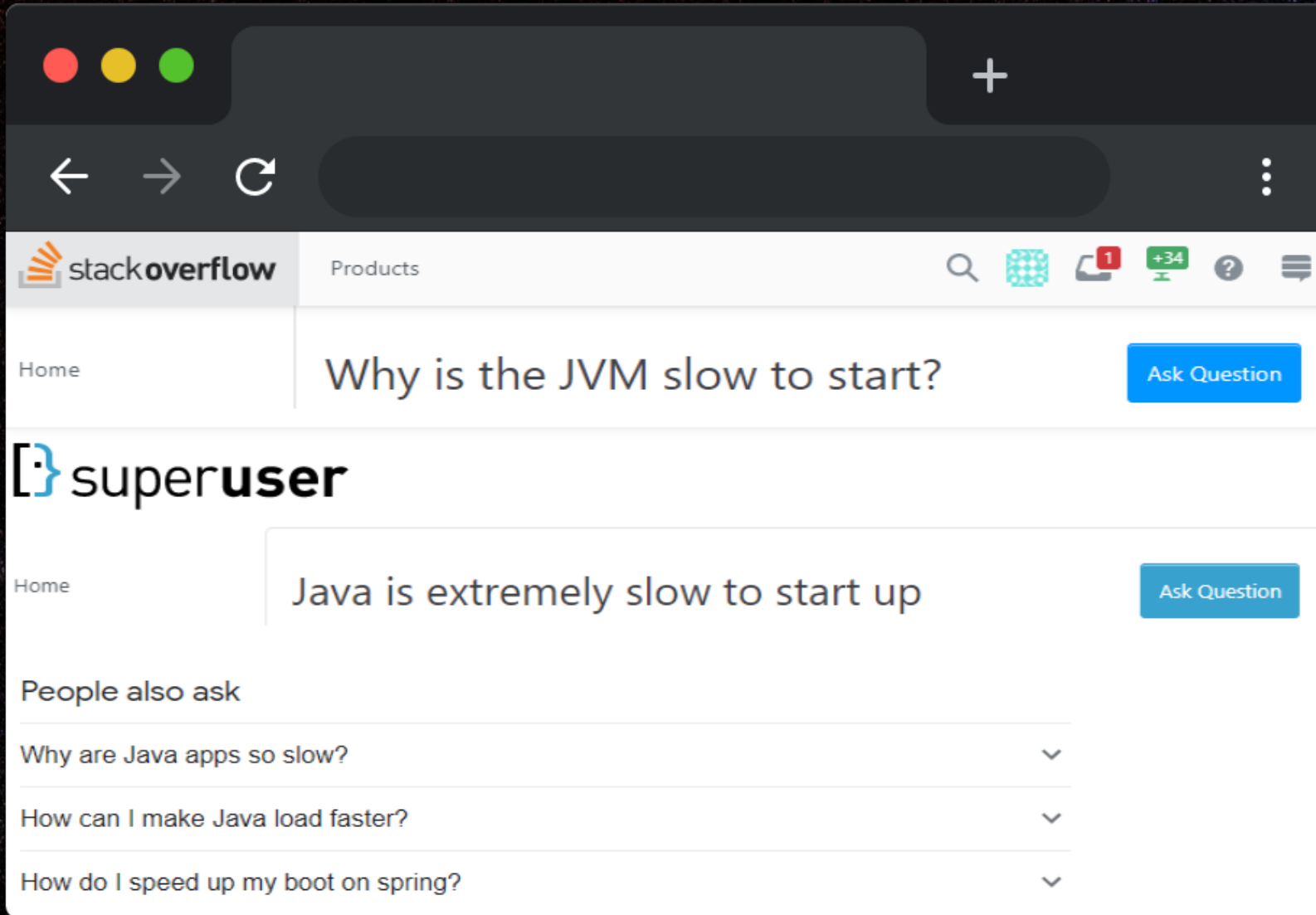
# About

## GraalVM™

## adesso

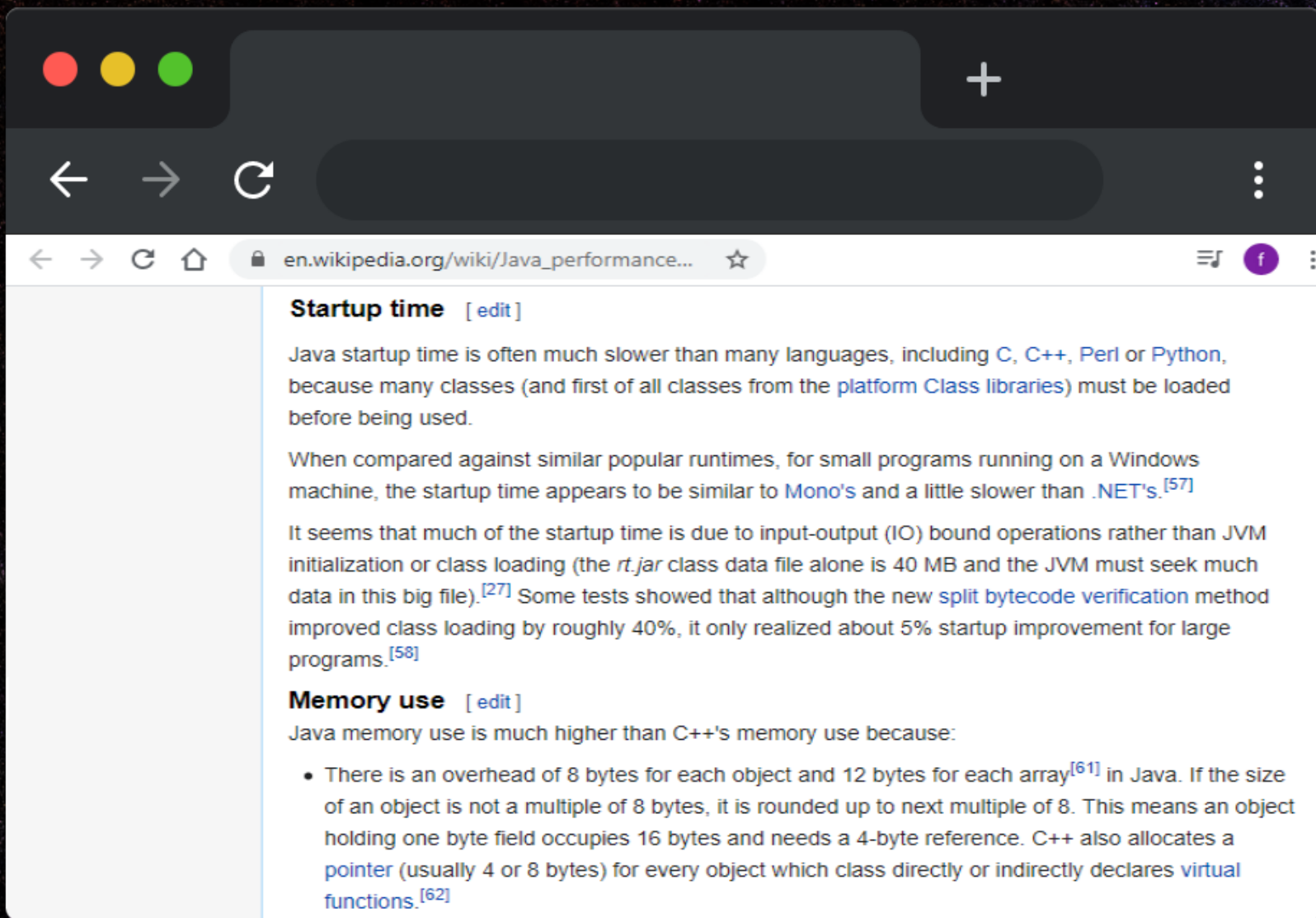


# The question is





# What we can do?



## Startup time [\[edit\]](#)

Java startup time is often much slower than many languages, including [C](#), [C++](#), [Perl](#) or [Python](#), because many classes (and first of all classes from the [platform Class libraries](#)) must be loaded before being used.

When compared against similar popular runtimes, for small programs running on a Windows machine, the startup time appears to be similar to [Mono's](#) and a little slower than [.NET's](#).<sup>[57]</sup>

It seems that much of the startup time is due to input-output (IO) bound operations rather than JVM initialization or class loading (the *rt.jar* class data file alone is 40 MB and the JVM must seek much data in this big file).<sup>[27]</sup> Some tests showed that although the new [split bytecode verification](#) method improved class loading by roughly 40%, it only realized about 5% startup improvement for large programs.<sup>[58]</sup>

## Memory use [\[edit\]](#)

Java memory use is much higher than C++'s memory use because:

- There is an overhead of 8 bytes for each object and 12 bytes for each array<sup>[61]</sup> in Java. If the size of an object is not a multiple of 8 bytes, it is rounded up to next multiple of 8. This means an object holding one byte field occupies 16 bytes and needs a 4-byte reference. C++ also allocates a [pointer](#) (usually 4 or 8 bytes) for every object which class directly or indirectly declares [virtual functions](#).<sup>[62]</sup>



# So what is the solution?

Wait.  
So you just save it,  
And your code is running?  
And it's Java?!



I know, right?  
SUPERSONIC JAVA, FTW!





# What is GraalVM ?

**GraalVM** is a **high-performance** JDK distribution designed to **accelerate the execution of applications** written in **Java** and other JVM languages along with support for **JavaScript**, **Ruby**, **Python** and ***a number of other popular languages***. **GraalVM**'s polyglot capabilities make it possible to mix multiple programming languages in a single application while ***eliminating foreign language call costs***.

**GraalVM** created by **Oracle**. The first production-ready version, **GraalVM 19.0**, was released in **May 2019**. There is no equivalent technology what **GraalVM** do.

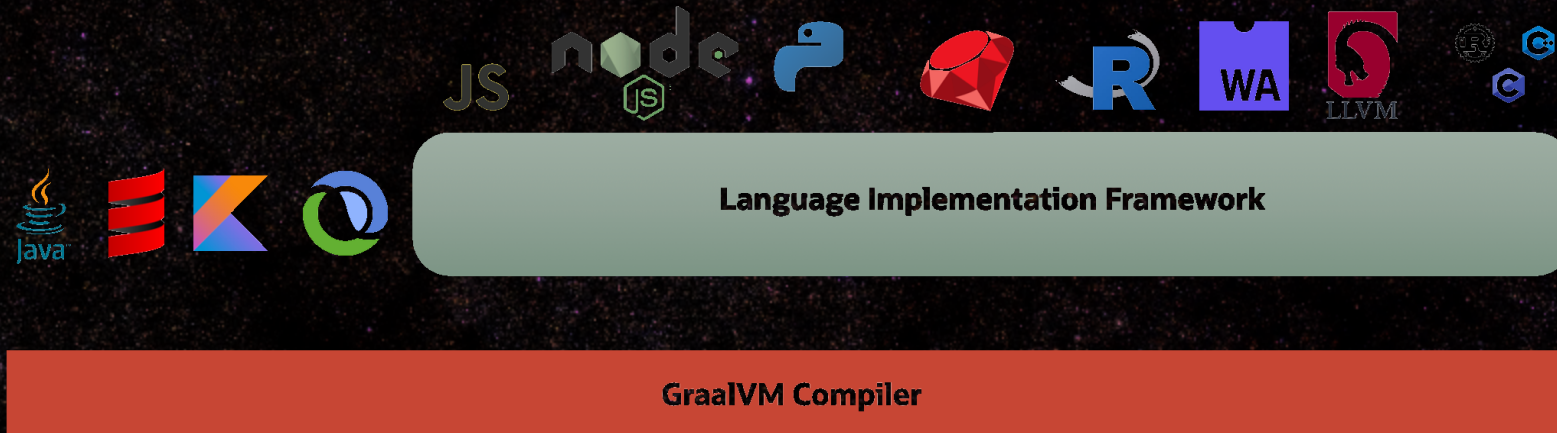


# What can GraalVM give me?

- Mix multiple programming languages in a single application
- Native executables
- Amazingly fast boot time
- Incredibly low RSS memory (not just heap size!)
- Instant (relatively) scale up and high density memory utilization in container orchestration platforms like Kubernetes.

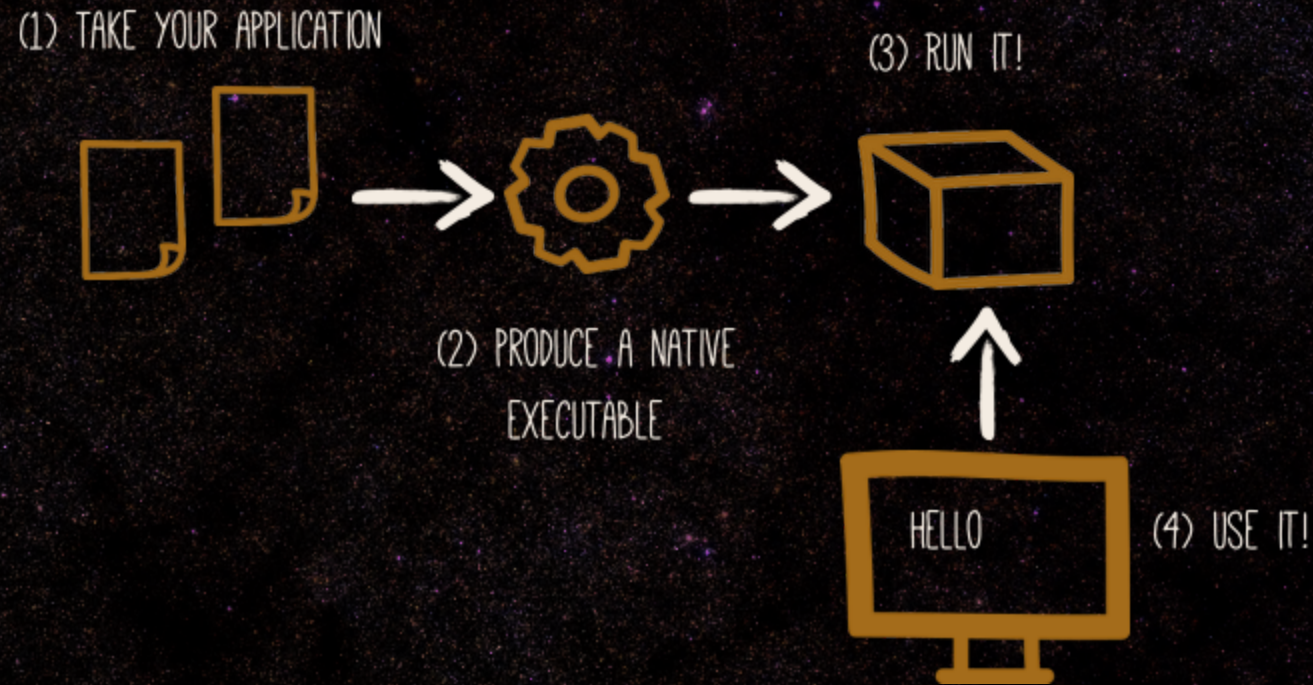


# GraalVM Architecture





# Native Image



The **native executable** for our application will contain the **application code**, **required libraries**, **Java APIs**, and **a reduced version of a VM**. The smaller VM base improves the **startup time** of the application and produces a **minimal disk footprint**.



# Truffle language implementation framework

The Truffle language implementation framework (henceforth “Truffle”) is an open source library for building tools and programming languages implementations as interpreters for self-modifying Abstract Syntax Trees. Together with the open source GraalVM compiler,

Truffle represents a significant step forward in programming language implementation technology in the current era of dynamic languages.



# Supported Frameworks and Tech

- Spring Framework (Experimental)
- Quarkus
- Play Framework
- Camel
- Prometheus
- JavaFX

...



# Disadvantages

- The main downside of this approach is the platform-dependent native code.

That means you need to compile source code for linux/windows etc.



# Meanwhile ??

**SystemAdmin01:** server\_api.so doesn't work on my server !

**Customer\$\$\$\_\_:** customer\_api.exe doesn't work on my pc !

**DevGuy42\_\_\_\_\_:** api.jar works on my workstation !

**F1rat\_\_\_\_\_:** In some cases, problems may arise from the libraries or build scripts used. In other words, it is necessary to pay attention to **tests**.

**SystemAdmin01:** himm, server\_api.so doest work on my server !

.....

**DevGuy42** writing ...



# Sample Application

Download and install GraalVM from oracle downloads. Create an `Example.java` file.

```
public class Example {  
  
    public static void main(String[] args) {  
        System.out.println("Hello GraalVM");  
    }  
}
```

```
#Run following commands :  
javac Example.java  
native-image Example
```



# Tests : Sure

```
package org.acme.quickstart;

import io.quarkus.test.junit.NativeImageTest;

@NativeImageTest
public class NativeGreetingResourceIT extends GreetingResourceTest {
    // Run the same tests
}
```

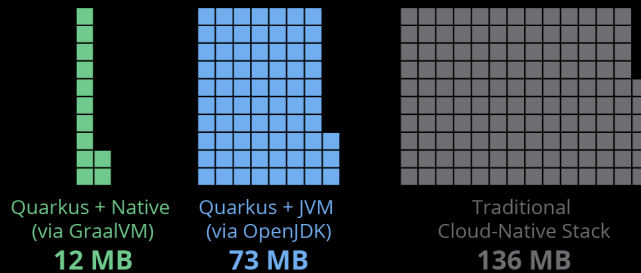


# Performance

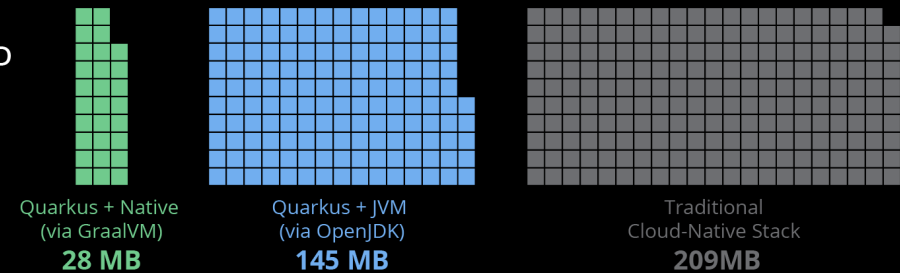
## Memory (RSS) in Megabytes\*

\*Tested on a single-core machine

REST



REST + CRUD



## BOOT + First Response Time

REST



REST + CRUD



Quarkus Test



# Features Support

GraalVM technologies are distributed as production-ready and experimental.

Experimental features are being considered for future versions of GraalVM and are not meant to be used in production. The development team welcomes feedback on experimental features, but users should be aware that experimental features might never be included in a final version, or might change significantly before being considered production-ready.



After that knowage i tried to make a demo application for my individual requirement.

Firstly i have prefer a desktop application with stylish UI.  
(relatively:))

That means the application sadly use a lot of memory. So i need to use `JavaFX` also `Gluon` (a native JavaFX solution) for improve to performance .



# What I did for graalvm integration?

For that integration i have only added a plugin to my maven pom.  
Surprisely that is enough for the integration.



```
<plugin>
  <groupId>com.gluonhq</groupId>
  <artifactId>gluonfx-maven-plugin</artifactId>
  <version>1.0.2</version>
  <configuration>
    <target>host</target>
    <mainClass>com.gnosis.cuteoverlay.Application</mainClass>
    <reflectionList>
      <list>com.gnosis.cuteoverlay.Application</list>
      <list>com.gnosis.cuteoverlay.DataToSend</list>
      <list>com.gnosis.cuteoverlay.DataToSendContainer</list>
    </reflectionList>
    <graalvmHome>C:\dev\tools\graalvm-svm-windows-gluon-21.2.0-dev</graalvmHome>
    <nativeImageArgs>
      <imageArg>-Dio.netty.noUnsafe=true</imageArg>
      <imageArg>--report-unsupported-elements-at-runtime</imageArg>
      <imageArg>--allow-incomplete-classpath --enable-all-security-services</imageArg>
      <imageArg>--enable-url-protocols=https -H:EnableURLProtocols=http</imageArg>
      <imageArg>-H:TraceClassInitialization=true</imageArg>
      <imageArg>--initialize-at-build-time=${buildtime-classes}</imageArg>
      <imageArg>--initialize-at-run-time=${runtime-classes}</imageArg>
    </nativeImageArgs>
  </configuration>
</plugin>
```



The main time spending problem is dedicating build and runtime classes also classes what used by Java's reflection. Also that lists depend on what libraries you used.

After several tries, the native image prepared successfully.

```
#creates native image  
mvn gluonfx:build
```

After that, i created a distribution folder than copied the native image and all other executables what i need to run business logic.



# And finally



Stockfish seviye 5

1 d4 e6  
2 c4 dxc6  
3 e3 h5  
4 dxc3 d5

firatgursoy 1500?

Sıra sizde

firatgursoy

CUTE OVERLAY		
CPU TEMP	55C	
GPU TEMP	46C	
CPU LOAD	07%	
GPU LOAD	00%	
MEM LOAD	72%	
GPU FAN	55%	



# Startup time metrics from my app

```
# Jar Log  
Uptime:781ms  
StartTime:Wed Jun 30 02:02:09 EET 2021  
Max heap memory is 4086 MBytes  
Used non-heap memory is 32 MBytes
```



```
# Native Image(exe) Log  
Uptime:1ms  
StartTime:Wed Jun 30 02:06:14 TRT 2021  
Max heap memory is 0 MBytes  
Used non-heap memory is 0 MBytes
```

Lightning fast jvm startup time !



# OS memory usages by the app

 CuteOverlay.exe	12520	Running	gno	00	109,928 K	Disabled
---	-------	---------	-----	----	-----------	----------

 Zulu Platform x64 Architecture	0%	180.4 MB	0 MB/s	0 Mbps
 Java(TM) Platform SE binary	0.4%	138.0 MB	0 MB/s	0 Mbps

We have 200MB saving from memory usage !



# Special thanks to

- Oracle, Quarkus, JavaFX / GluonHQ, Spring
- Google and Wiki



You can reach the source code and windows builds via git.

<https://github.com/firatgursoy/CuteOverlay>