locking.

2PL two-phase locking

acquire ALL locks
⋮
release ALL locks

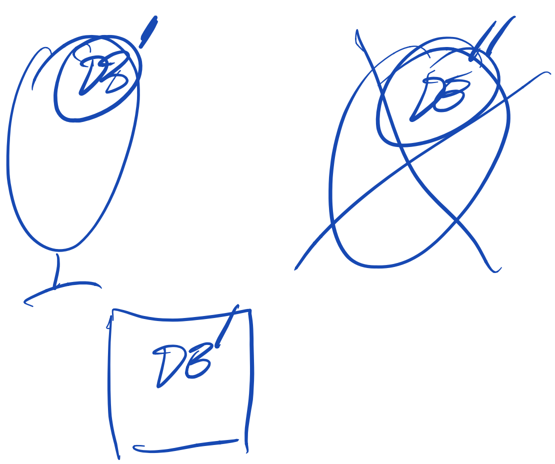
guarantees atomicity

$\alpha \to \beta \to \gamma \qquad \beta \to \gamma \qquad \alpha \to \gamma$

pessimism vs. optimism

canonical lock ordering $\Rightarrow$ no deadlock (no cycles)

lkml.org/lkml/2017/10/4/580



OCC:

conflicts $\Rightarrow$ aborts

[correctness vs. performance
ease-of-programming " "]

high level PL — SQL
Haskell
Scala

↑ ease of programming
↑ correctness
↓ perf.

low level abstraction

C/C++

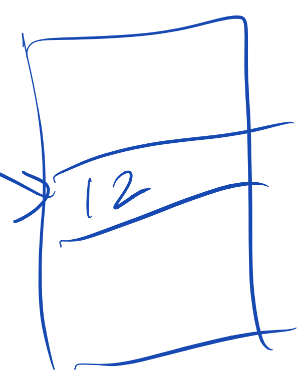
Java
C/C++

"safe PL"

memory

C++

```
auto v = 39612548;
~~void~~ int * p = (~~void~~ int *) v;
*p = 12;
```



SEGV

Cannot do this in Java, Scala, Haskell, ...

~~fault~~ → score crash

SQL? not "Turing-complete"

ANSI SQL92 — Stored procedures extensions

Java

Sun

HotJava browser

applet

~~ActiveX~~

Interpreted

GC

Java

C#

(.NET)

"UDO"

User-defined operator

Brendan Eich

↪ JavaScript

NSPR

Netscape Portable Runtime

Java
- rich set of libraries
- run "anywhere"

WORA

Server land

~~Sun~~ Oracle

JVM

Java Virtual Machine

...tion set

pseudo-instruct...

JVM bytecodes

stack-oriented ISA

javac file.java

→ file.class

push a
push b
ADD

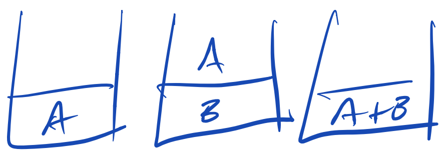
register-oriented ISA

Dalvik → Dalvik VM bytecodes

(Android)

Hoard

DieHard



MOV A, R1
MOV B, R2
ADD R1, R2, R3

Hadoop, Spark ...

Java / JVM language

(Scala, ...
Clojure
....)

C/C++

- direct access to memory } (raw) unsafe memory access
- new / delete
  ~~free~~
  malloc/free } explicit mem mgmt
- compiled ahead-of-time

-O3
clang++ foo.cpp -o foo

→ foo

x86/ARM ...

```
Foo * f = new Foo;
Foo * g = f;
delete f;
Bar * b = new Bar;
g -> printMe();
```

Use after free

Java

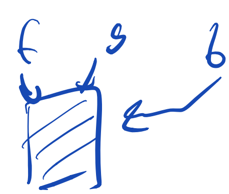
- indirect access to memory
- bounds checks
- no explicit addresses
  "references"
  (opaque)

- write
- read

code executed ⇒ "barrier"

new
garbage collection
+ safety
- costs

JIT compiler



Just-in time

"HotSpot"

JIT: interp
↓
...

dangling pointer error

GC to the rescue!

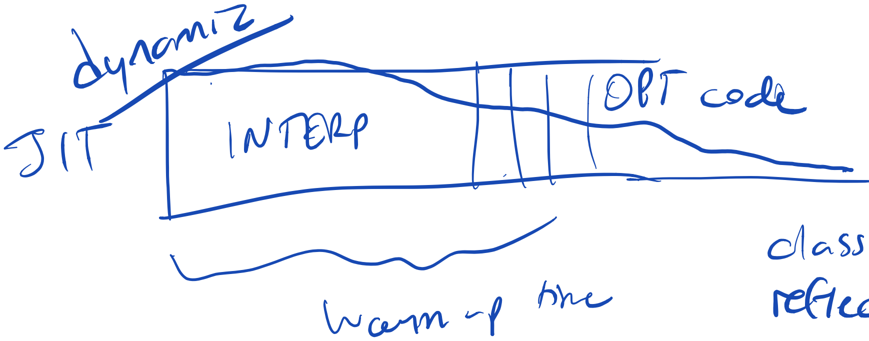
O1
↓
O2
↓
O3

```
Foo f = new Foo();
Foo g = f;
Bar b = new Bar();
g.printMe();
f = null;
g = null;
```

reachability
⇒ not collected

AOT – eager   ahead of time

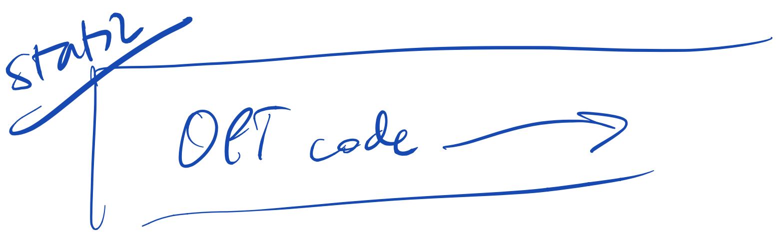
JIT – lazy   on demand



throughput focused

class loaders
reflection

"eval"



latency
& throughput
focused

— bounds checks

$x[i]$

is $i < 0$?
is $i > \text{bound}$? ⟹ exception

for ($i = 0$; $i < 10$; i++)
  $z = 12 * 15$;
  ... $x[i]$ ...

HOISTING

~ Can be "compiled out" ~

INLINING

f() {
  A
  g();
  E
}

g() {
  B
  h();
  D
}

h() {
  C
}

⟹ f() {
  A;
  B;
  C;
  D;
  E;
}

~~$x = 1$~~
~~$x = 2$~~
~~$x = x + 1$~~
$x = x + 2$
$x = \cancel{x} * x$

~~$x = 3$~~
~~$x = 5$~~
$x = 25$

Constant propagation

Copy propagation

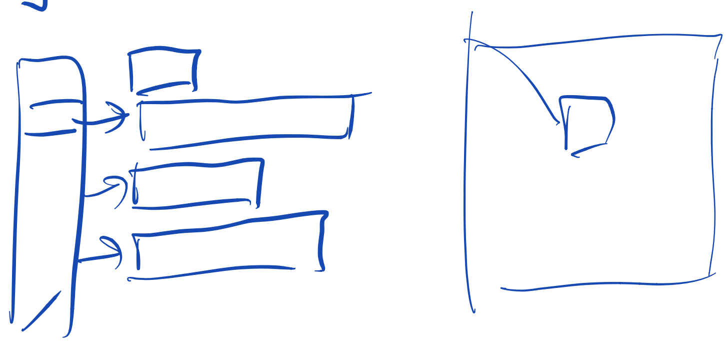
⋮

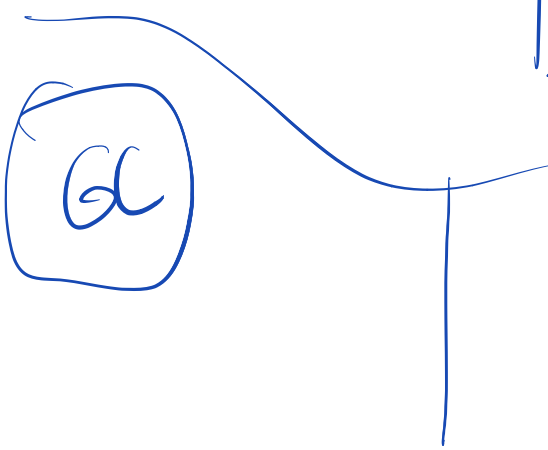
inlining exposes opt. opportunities

CODE QUALITY overhead ⇒ 5-15% slowdown Java over C

(not Java matrices)

a[i][j]





C/C++

low-level ⇒ "more expressive"

Cache-aware

GC: unpredictable latency

JVM ⟶ x86

"hard!"

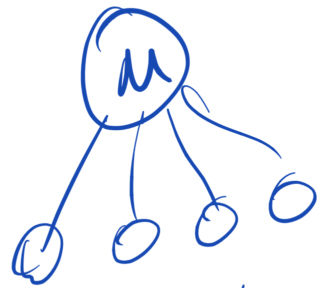
⟶ STRAGGLER PROBLEM

→ SPACE

vs.

TIME

GC trades space for time



barrier