JMM
JIT
GC

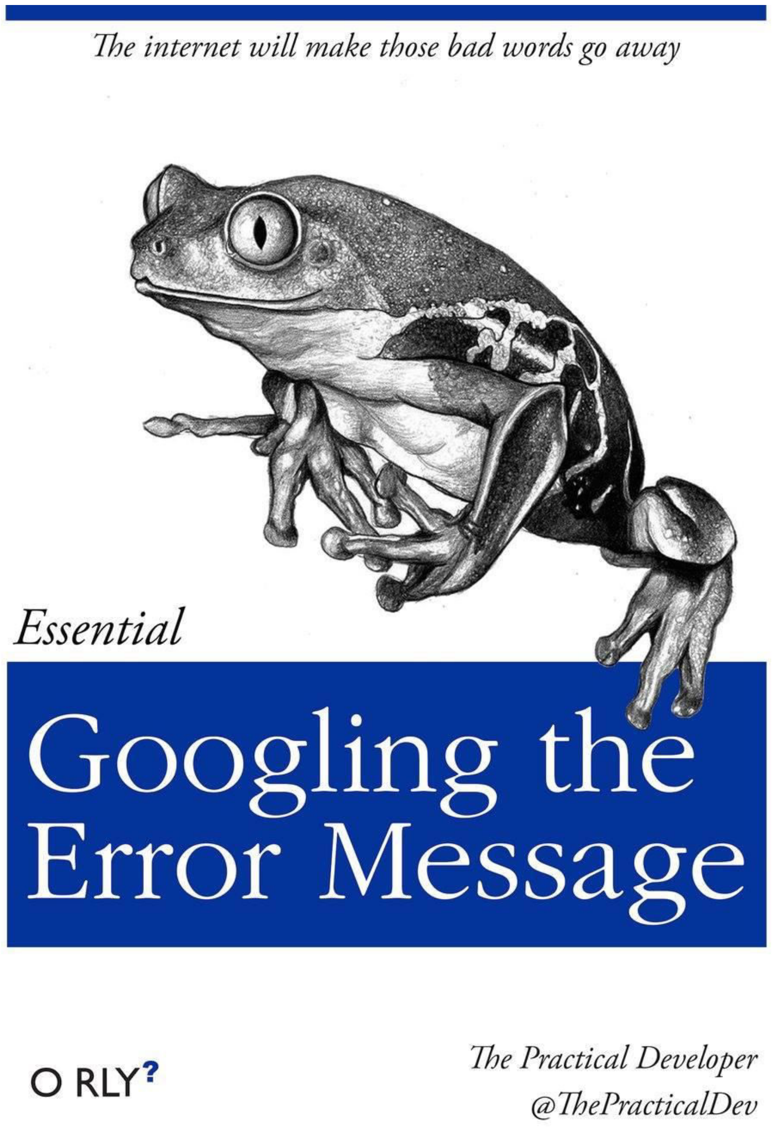
Rikhard Goldenberg
SberTech JavaSchool, Mar.2018

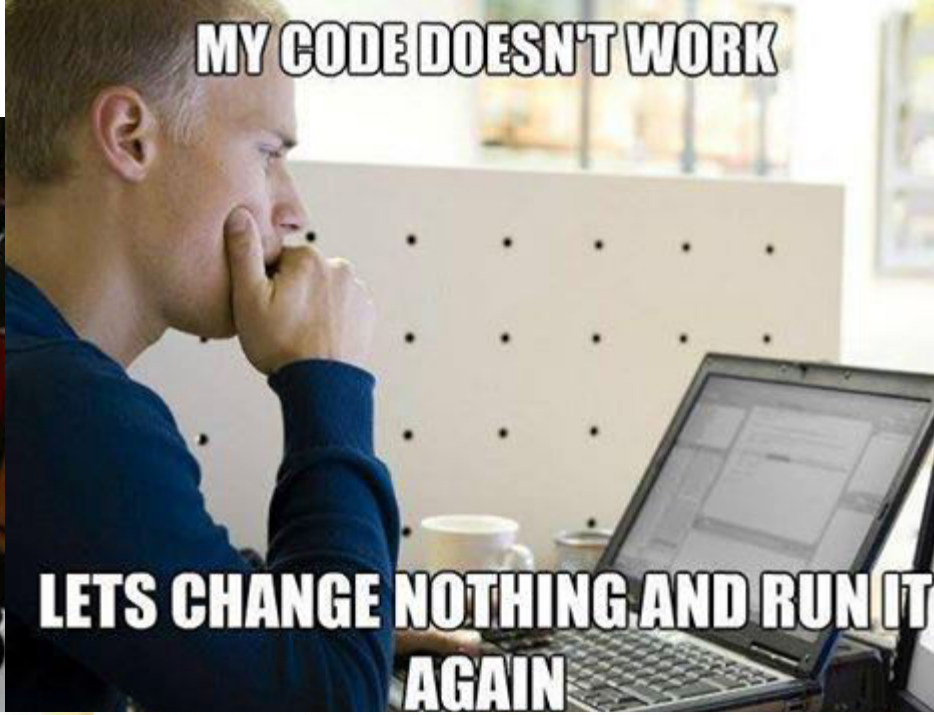
JAVA DEVELOPER

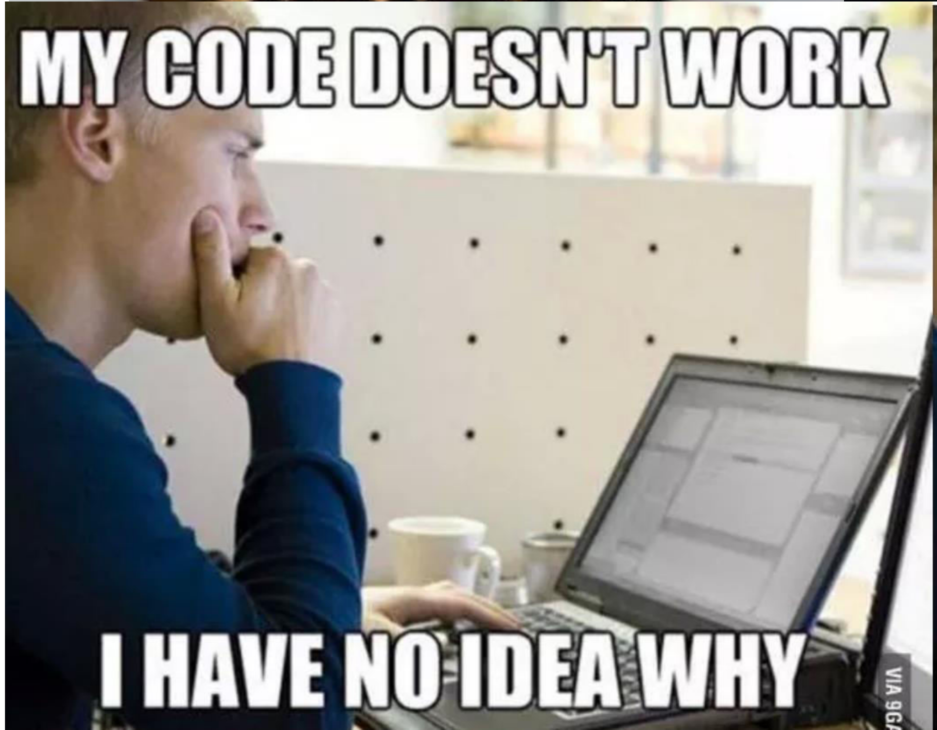
VS

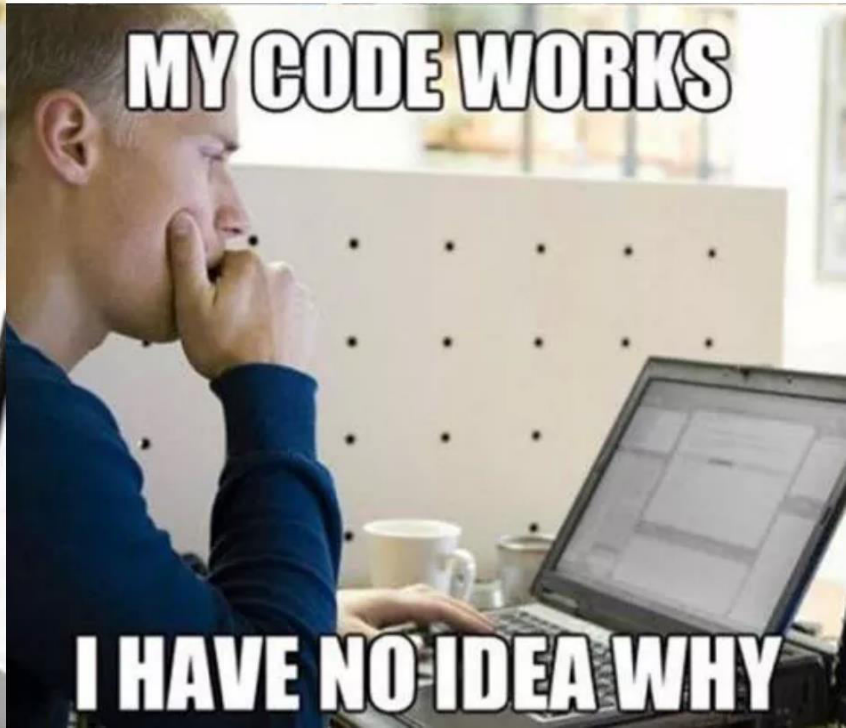
JAVA DEVELOPER

JAVA DEVELOPER

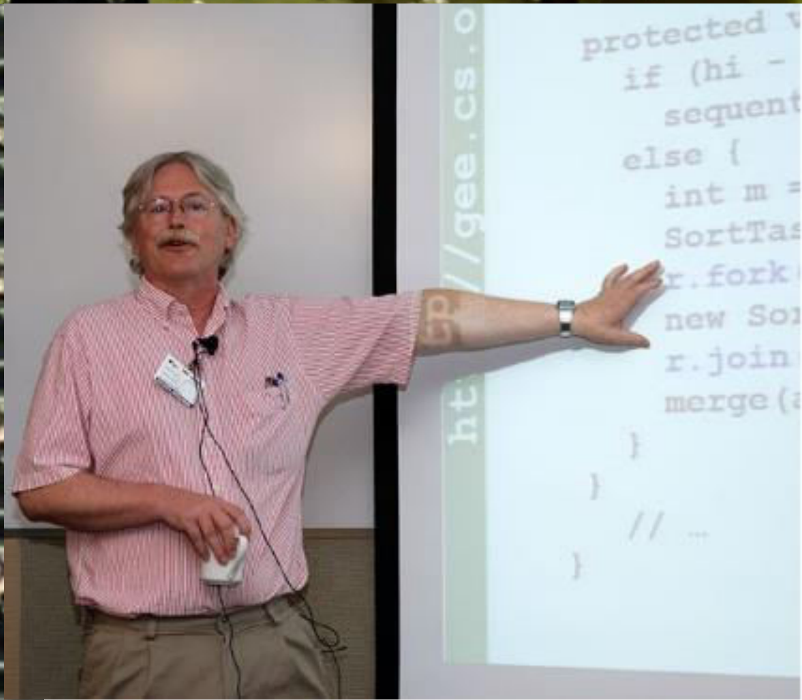








JAVA DEVELOPER

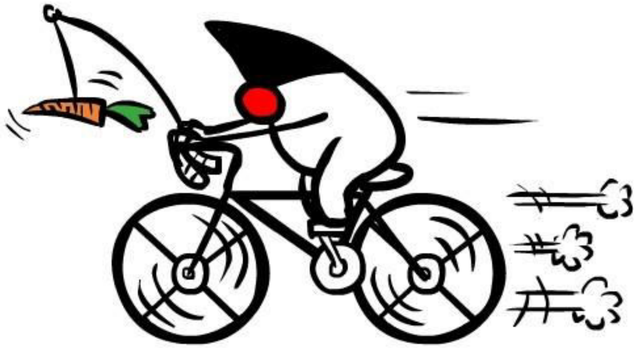


- JMM
- JIT
- GC

JAVA MEMORY MODEL

1980s – one CPU instruction execution == one RAM access action

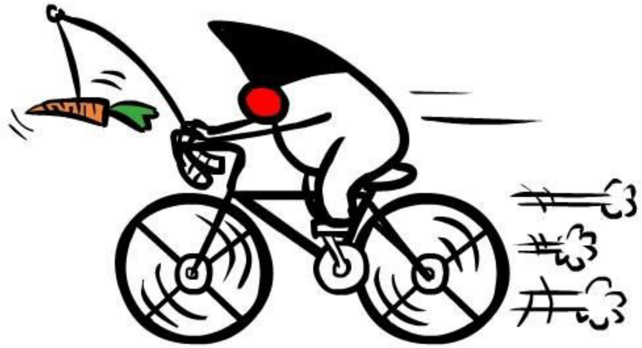
2010s – CPU performance increase - ~10_000 times
RAM performance increase - ~10 times



1 CPU cycle	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	2.8 ns
Level 3 cache access	12.9 ns
Main memory access	120 ns
Solid-state disk I/O	50-150 μs
Rotational disk I/O	1-10 ms
Internet: SF to NYC	40 ms
Internet: SF to UK	81 ms
Internet: SF to Australia	183 ms

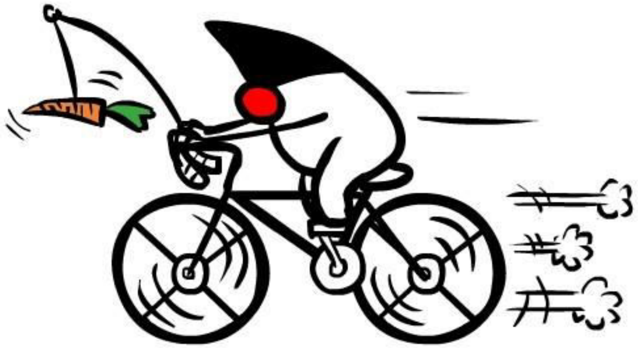
1980s – one CPU instruction execution == one RAM access action

2010s – CPU performance increase - ~10_000 times
RAM performance increase - ~10 times



1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

HEAVY OPTIMIZATION NEEDED



1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

```
int foo = 0;
int bar = 0;

void run {
      foo += 1;
      bar += 1;

      foo += 2;
}
```

REORDERING

```
int foo = 0;
int bar = 0;

void run {
      foo += 1;
      bar += 1;

      foo += 2;
}


void run {
      foo += 1;
      foo += 2;

      bar += 1;
}
```

REORDERING

```
int foo = 0;
int bar = 0;

void run {
      foo += 1;
      bar += 1;

      foo += 2;
}
```

foo == 0, bar == 0
foo == 1, bar == 0
foo == 1, bar == 1
foo == 3, bar == 1

```
void run {
      foo += 1;
      foo += 2;

      bar += 1;
}
```

foo == 0, bar == 0
foo == 1, bar == 0
foo == 3, bar == 0
foo == 3, bar == 1

СБЕРБАНК ТЕХНОЛОГИИ

REORDERING

```
int foo = 0;
int bar = 0;

void run {
      foo += 1;
      bar += 1;

      foo += 2;
}
```

foo == 0, bar == 0
foo == 1, bar == 0
foo == 1, bar == 1
foo == 3, bar == 1

```
void run {
      foo += 3;
      bar += 1;
}
```

foo == 0, bar == 0
???
foo == 3, bar == 0
foo == 3, bar == 1

```
boolean flag = true;
int count = 0;
```

```
void run { //thread1
    while (flag) {
        count++;
    }
}
```

```
void run { //thread2
    flag = false;
}
```

```
boolean flag = true;
int count = 0;
```

JIT OPTIMIZATION

```
void run { //thread1
      while (flag) {
            count++;
      }
}
```

```
void run { //thread2
      flag = false;
}
```

```
void run { //thread1
      while (true) {
            count++;
      }
}
```

```
boolean flag = true;
int count = 0;
```

JIT OPTIMIZATION

```
void run { //thread1
    while (flag) {
        count++;
    }
}
```

```
void run { //thread2
    flag = false;
}
```

```
void run { //thread1
    while (true) {
        count++;
    }
}
```

```
void run { //thread2
    //flag = false;
}
```

JAVA MEMORY MODEL

answers the question:
“What values can we observe upon reading from a specific field?”

СБЕРБАНК ТЕХНОЛОГИИ

JAVA MEMORY MODEL

answers the question:
“What values can we observe upon reading from a specific field?”

```
int foo = 0;

//single-threaded
void run {
     foo = 1;
     if (foo < 1) {
          throw new ISE();
     }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

JAVA MEMORY MODEL

answers the question:
"What values can we observe upon reading from a specific field?"

```
int foo = 0;

//single-threaded
void run {
    foo = 1;          //->  1.
    if (foo < 1) {    //->  2.
        throw new ISE();
    }
}
```

program **order**

JAVA MEMORY MODEL

answers the question:
"What values can we observe upon reading from a specific field?"

```
int foo = 0;
int bar = 0;
//single-threaded
void run {
      //bar = 1;
      foo = 1;
      //bar = 1;
      if (foo < 1) {
            throw new ISE();
      }
      //bar = 1;
}
```

JAVA MEMORY MODEL

volatile
synchronized
java.util.concurrent.*

СБЕРБАНК ТЕХНОЛОГИИ

```
int foo = 0;
int bar = 0;
int baz = 0;

//single-threaded
void run {
     foo += 1;
     bar += 1;
     baz += 1;
}
```

```
volatile
```

volatile

```
int foo = 0;
int bar = 0;
int baz = 0;

//single-threaded
void run {
      foo += 1;
      bar += 1;
      baz += 1;
}
```

```
void run {
        baz += 1;
        foo += 1;
        bar += 1;
}
void run {
        foo += 1;
        baz += 1;
        bar += 1;
}
void run {
        bar += 1;
        foo += 1;
        baz += 1;
}
```

volatile

```
int foo = 0;
volatile int bar = 0;
int baz = 0;

//single-threaded
void run {
      foo += 1;
      bar += 1;    //-> PROGRAM ORDER BARRIER
      baz += 1;
}
```

СБЕРБАНК ТЕХНОЛОГИИ

volatile

```
int foo = 0;
volatile int bar = 0;
int baz = 0;
```

```
//multi-threaded
void run {//thread1
      foo += 1;
      bar += 1;
      baz += 1;
}
```

happens-before

```
//multi-threaded
void run {//thread2
      foo += 1;
      bar += 1;
      baz += 1;
}
```

time

JAVA MEMORY MODEL

"happens-before" order

- **action in a thread *happens-before* any subsequent action in that thread**
- **writing to a volatile variable *happens-before* reading from same variable**
- **thread.start() *happens-before* any action in that thread**
- **all actions in a thread *happen-before* a return from a call to .join() on that thread**
- **monitor release *happens-before* any subsequent monitor acquisition on the same object**

Brian Goetz
with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea

Java Concurrency in Practice

Chapter 17

- JMM
- JIT
- GC

JUST-IN-TIME COMPILATION

СБЕРБАНК ТЕХНОЛОГИИ

AOT -> Ahead-Of-Time compilation happens prior to execution

JIT -> Just-In-Time compilation happens during execution

AOT

- **Pros**
  - can perform time-consuming analysis and complex optimizations
- **Cons**
  - static information in not always enough to optimize effectively
  - cannot use profiling
  - cannot make use of previously unavailable hardware features

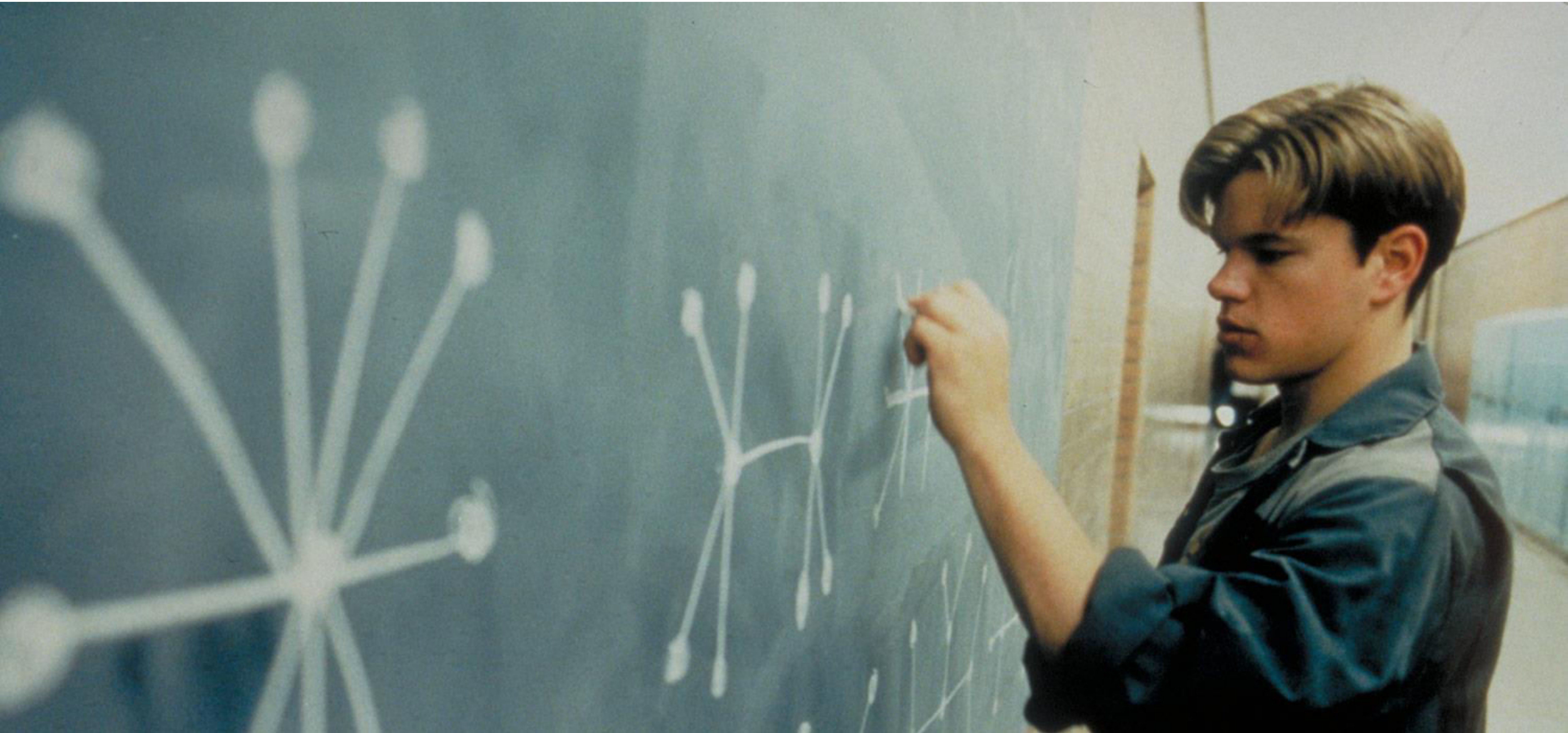
JIT

- **Pros**
  - uses aggressive speculative optimizations based on profiling information
  - can use new hardware features out-of-the-box
- **Cons**
  - uses resources at runtime
  - can negatively affect startup time
  - peak performance can be lower

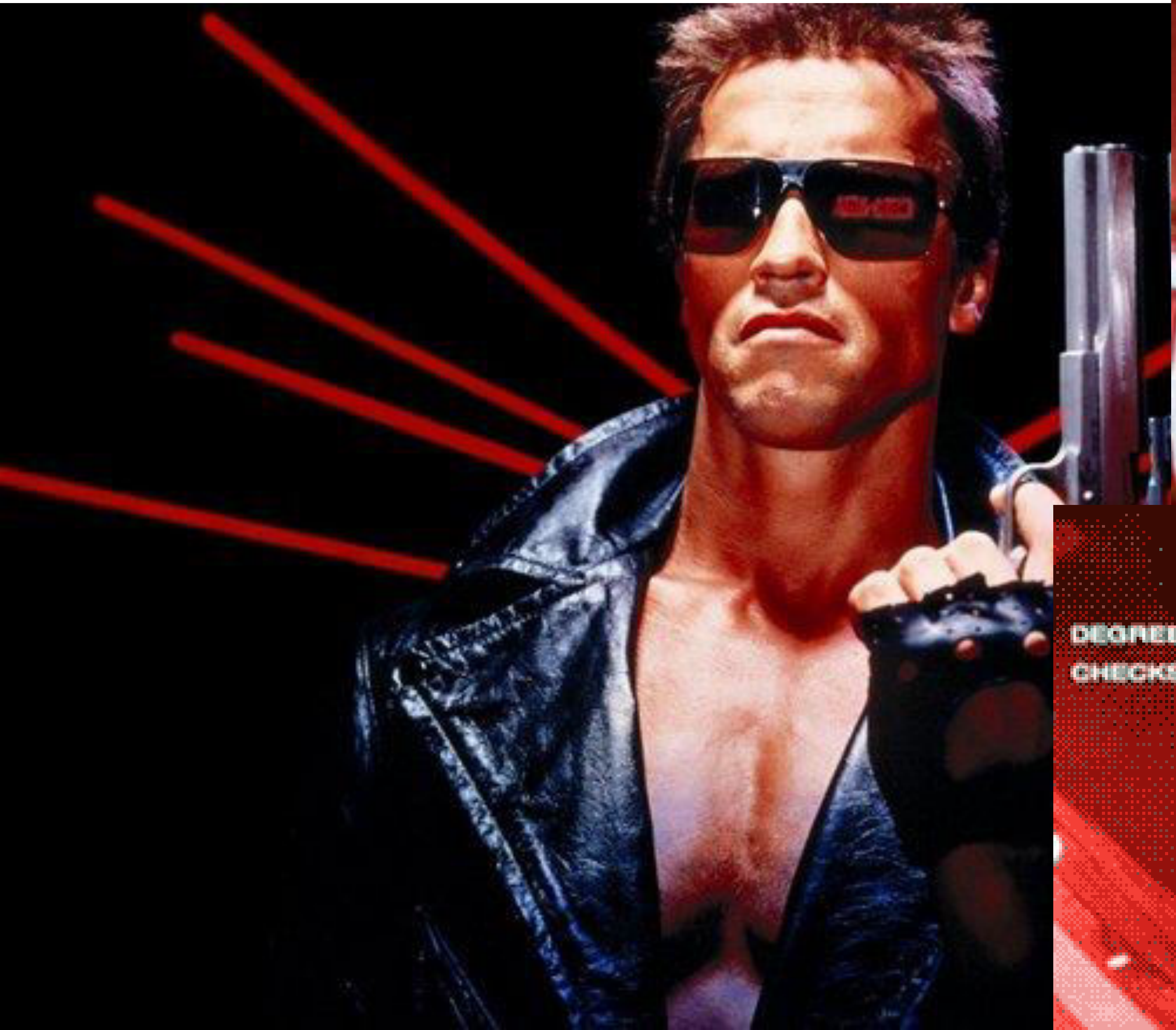
JIT

СБЕРБАНК ТЕХНОЛОГИИ

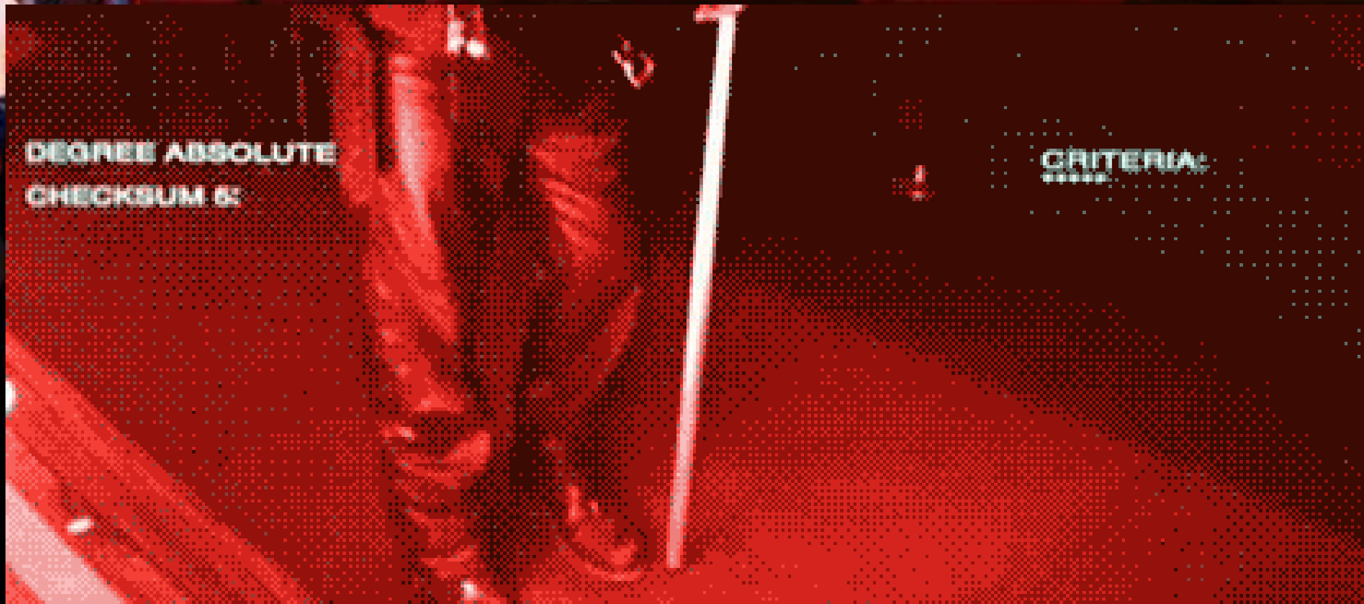
AOT



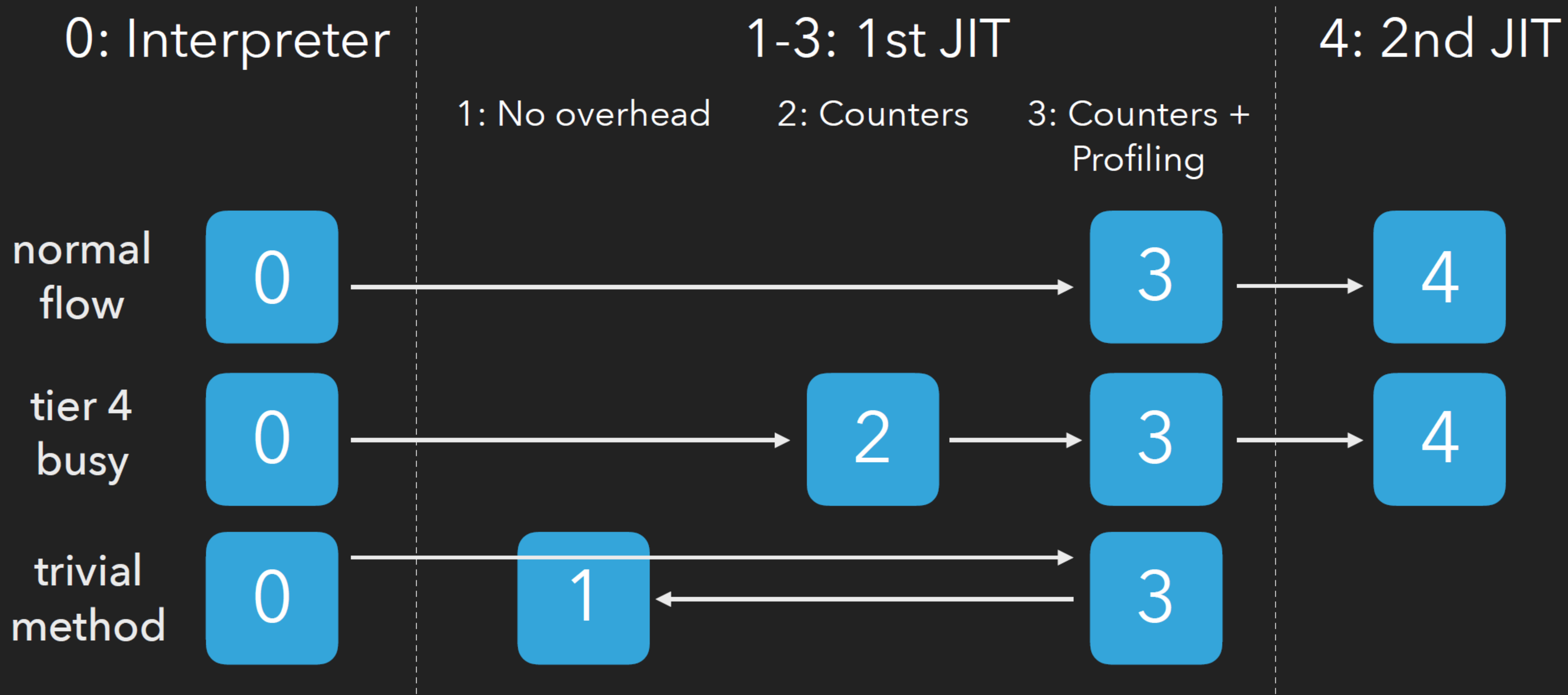
JIT



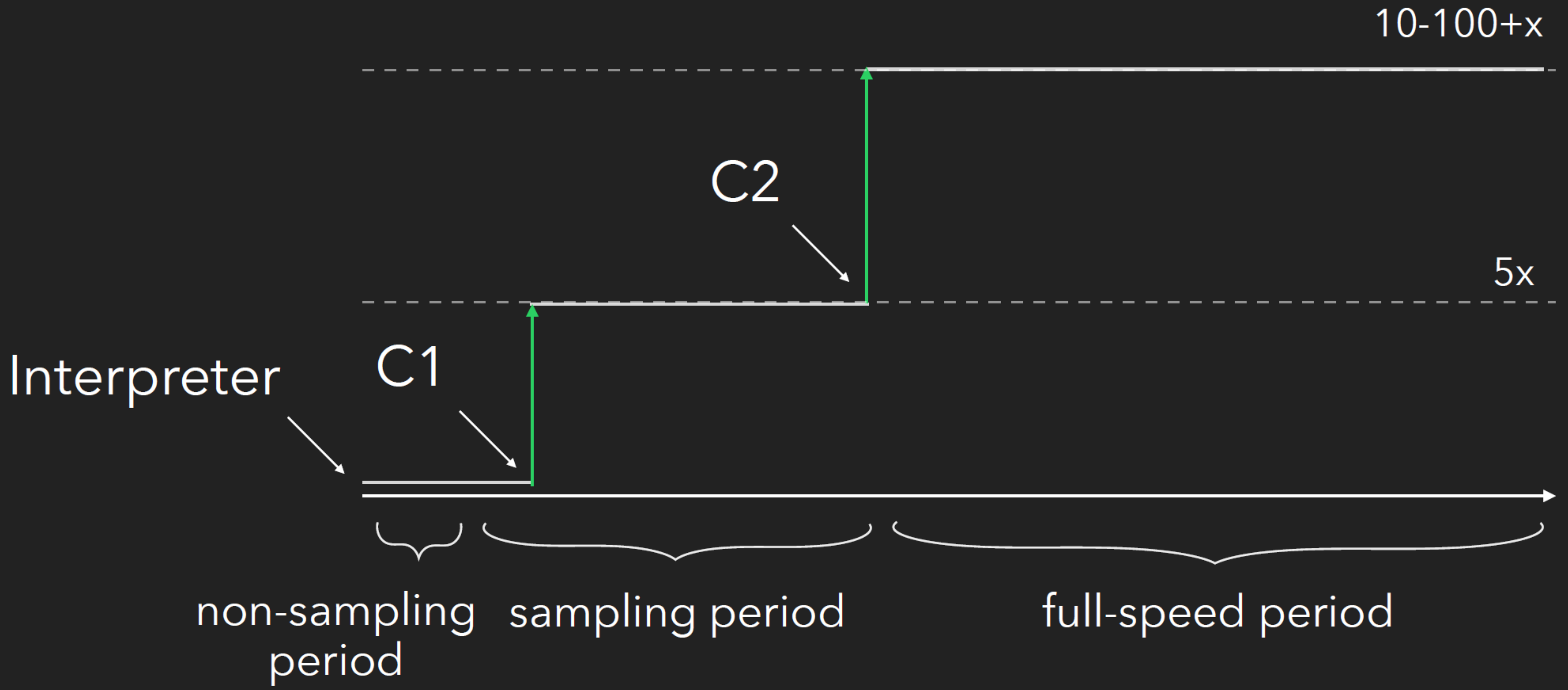

5439 543 5435 65311
6465 656 7689 10930
SCAN
AXIL
CAMI
CHKW
EPIG
FDSK
MAND
MAST
MAXI
MO
HUMAN CASUALTIES:


DEGREE ABSOLUTE
CHECKSUM 6:
CRITERIA:

TIERED COMPILATION



MENTAL MODEL



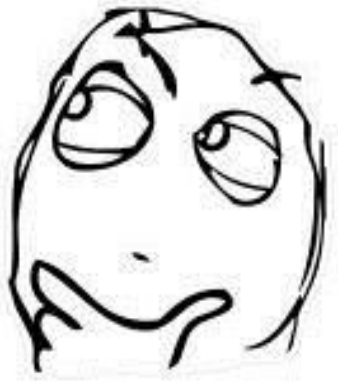
СБЕРБАНК ТЕХНОЛОГИИ

JIT OPTIMIZATIONS

- compiler tactics
  - delayed compilation
  - tiered compilation
  - on-stack replacement
  - delayed reoptimization
  - program dependence graph representation
  - static single assignment representation
- proof-based techniques
  - exact type inference
  - memory value inference
  - memory value tracking
  - constant folding
  - reassociation
  - operator strength reduction
  - null check elimination
  - type test strength reduction
  - type test elimination
  - algebraic simplification
  - common subexpression elimination
  - integer range typing
- flow-sensitive rewrites
  - conditional constant propagation
  - dominating test detection
  - flow-carried type narrowing
  - dead code elimination
- language-specific techniques
  - class hierarchy analysis
  - devirtualization
  - symbolic constant propagation
  - autobox elimination
  - escape analysis
  - lock elision
  - lock fusion
  - de-reflection
- speculative (profile-based) techniques
  - optimistic nullness assertions
  - optimistic type assertions
  - optimistic type strengthening
  - optimistic array length strengthening
  - untaken branch pruning
  - optimistic N-morphic inlining
  - branch frequency prediction
  - call frequency prediction
- memory and placement transformation
  - expression hoisting
  - expression sinking
  - redundant store elimination
  - adjacent store fusion
  - card-mark elimination
  - merge-point splitting
- loop transformations
  - loop unrolling
  - loop peeling
  - safepoint elimination
  - iteration range splitting
  - range check elimination
  - loop vectorization
- global code shaping
  - inlining (graph integration)
  - global code motion
  - heat-based code layout
  - switch balancing
  - throw inlining
- control flow graph transformation
  - local code scheduling
  - local code bundling
  - delay slot filling
  - graph-coloring register allocation
  - linear scan register allocation
  - live range splitting
  - copy coalescing
  - constant splitting
  - copy removal
  - address mode matching
  - instruction peepholing
  - DFA-based code generator

```
void run {
      int number = 2;
      for (int i = 0; i < 10; i++) {
            number = addSelf(number);
      }
}


int addSelf(int num) {
      return num + num;
}
```



СБЕРБАНК ТЕХНОЛОГИИ

```
void run {
    int number = 2;
    for (int i = 0; i < 10; i++) {
        number = number + number;
    }
}
```

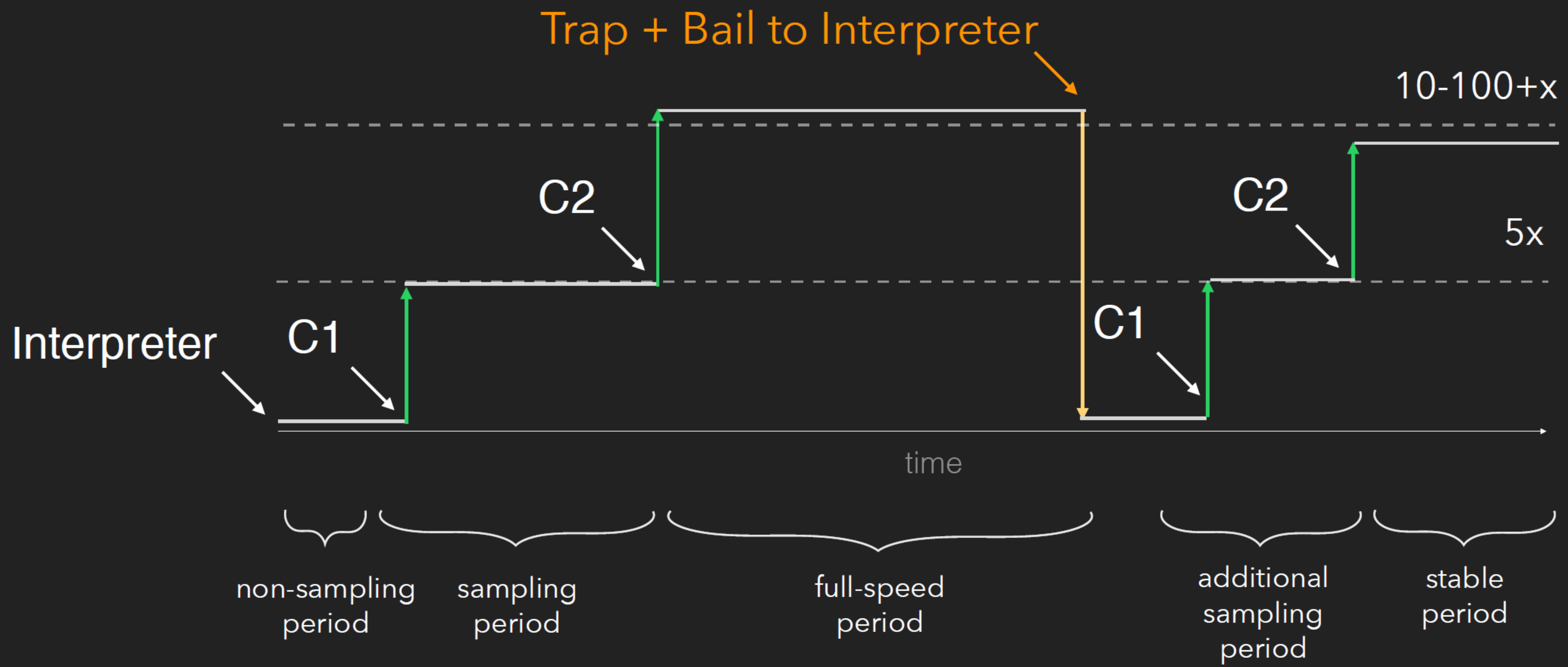
OPENS WAY FOR COUNTLESS OTHER OPTIMIZATIONS

METHODS

- Monomorphic
- Bimorphic
- Polymorphic
- Megamorphic

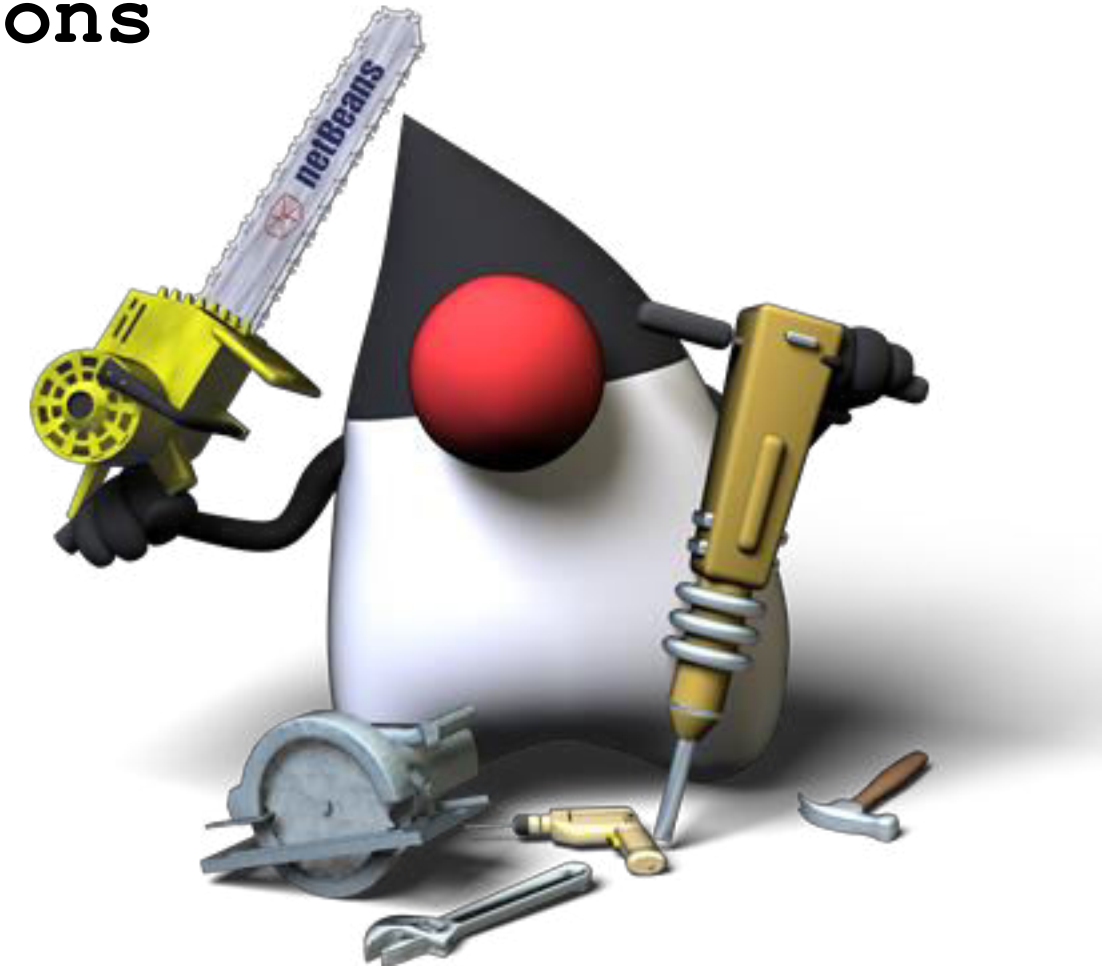
DEOPTIMIZATION

REVISED MENTAL MODEL



```
-XX:+PrintCompilation

-XX:+UnlockDiagnosticVMOptions
-XX:+PrintInlining
```



СБЕРБАНК ТЕХНОЛОГИИ

- JMM
- JIT
- GC

GARBAGE COLLECTION

СБЕРБАНК ТЕХНОЛОГИИ

GARBAGE COLLECTION

parallel vs concurrent

concurrent vs stop-the-world

monolithic vs incremental

СБЕРБАНК ТЕХНОЛОГИИ

GARBAGE COLLECTION

parallel vs concurrent

concurrent vs stop-the-world

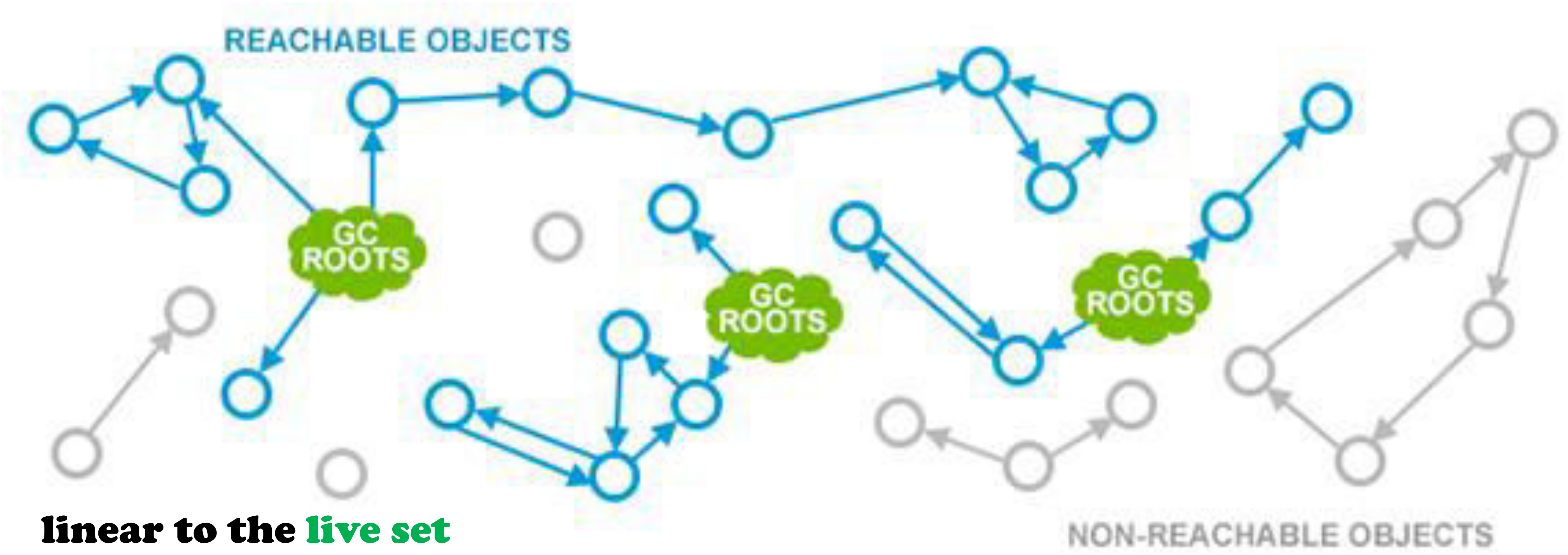
monolithic vs incremental

(GC) SAFEPOINTS

LIVE SET

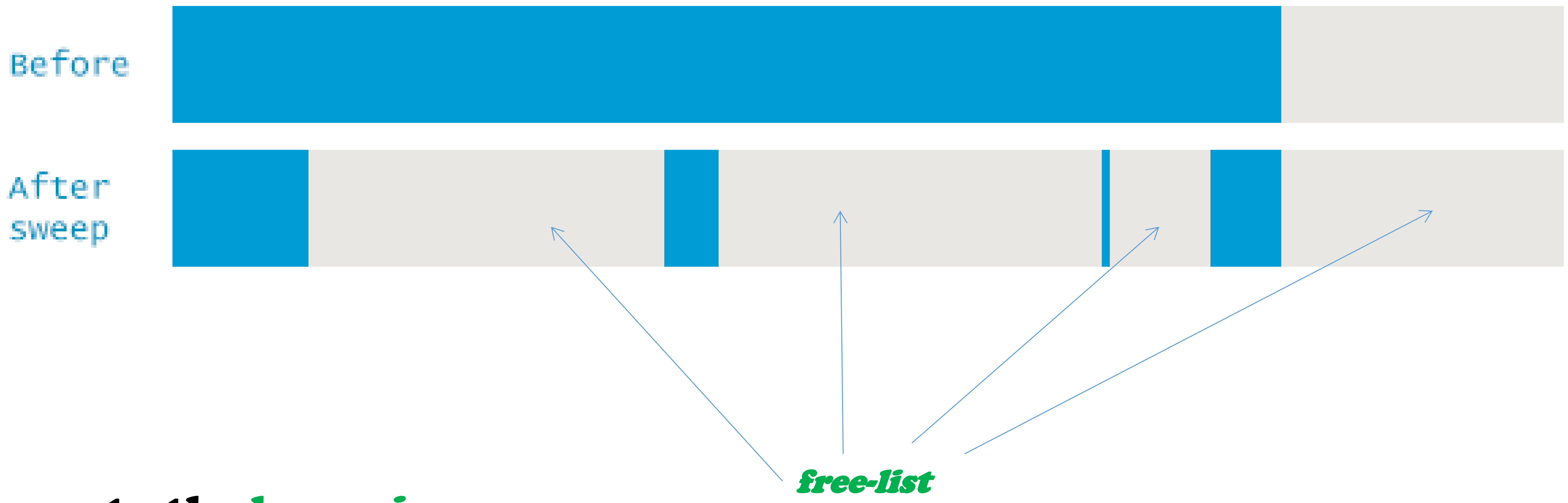
СБЕРБАНК ТЕХНОЛОГИИ

MARK



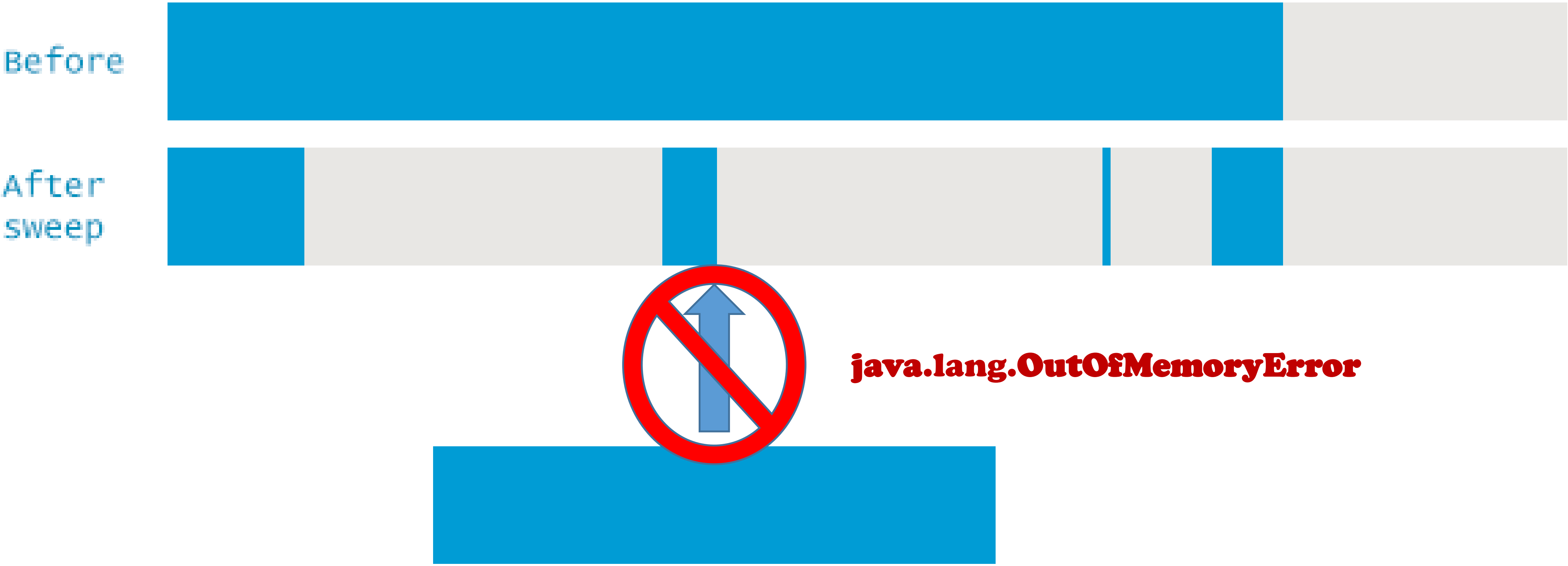
linear to the live set

SWEEP



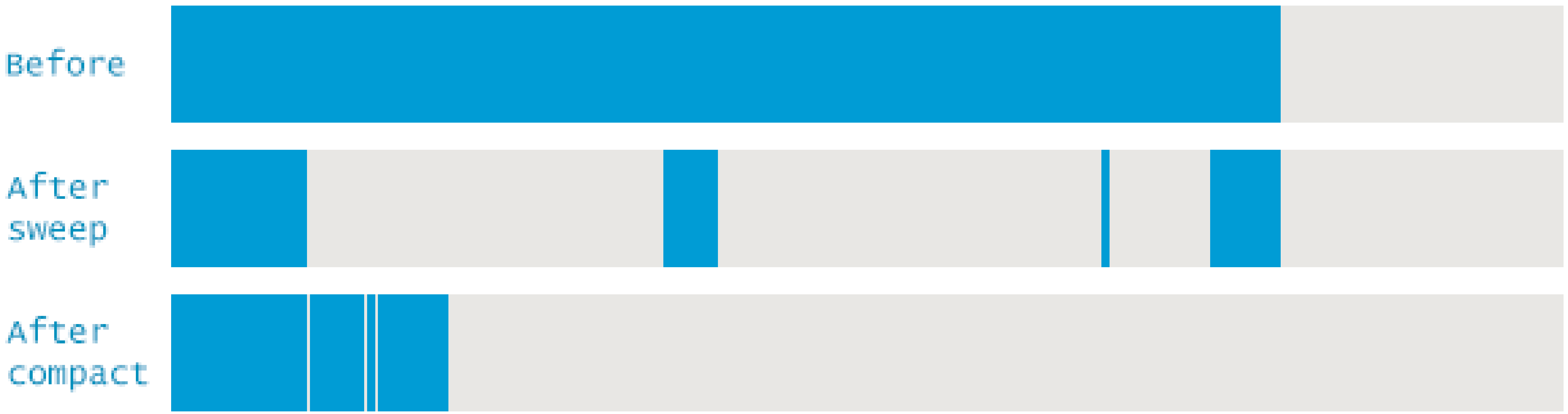
linear to the heap size

SWEEP



MARK/SWEEP/COMPACT

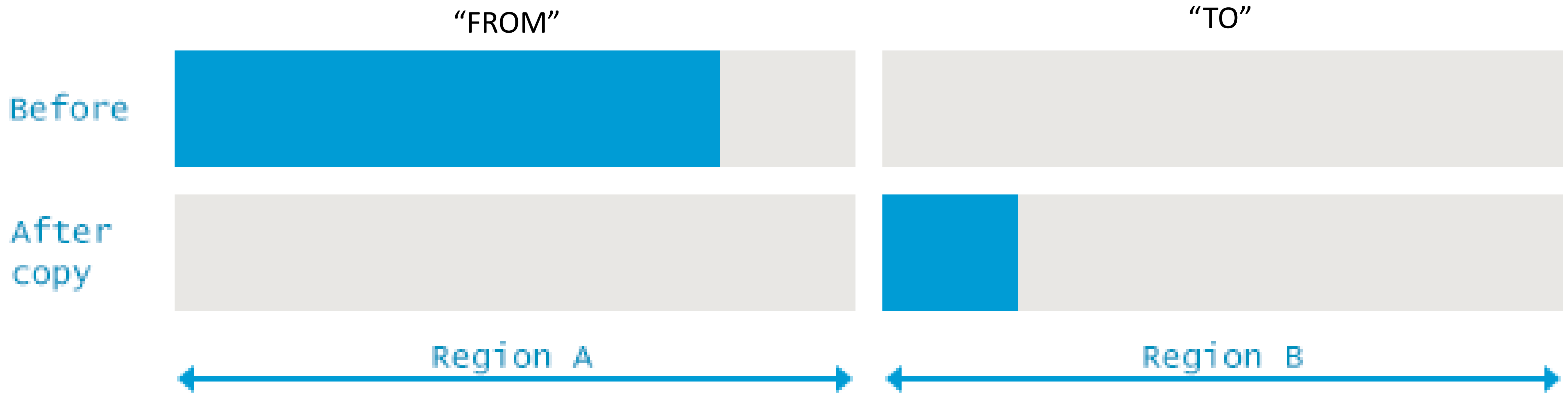
relocate -> remap



linear to the live set

СБЕРБАНК ТЕХНОЛОГИИ

(MARK +) COPY



linear to the live set; single pass

requires 2x live set memory

GARBAGE COLLECTION

mark-sweep-compact, copy, mark-compact, mark-sweep-(compact)

- ***COPY***
  - ***needs 2x live set memory***
  - ***linear to live set***
  - ***Monolithic***

- ***MARK-COMPACT***
  - ***needs 2x live set memory***
  - ***linear to live set***

- ***MARK-SWEEP-COMPACT***
  - ***1x live set memory***
  - ***linear to heap size (during sweep)***

- ***MARK-SWEEP-(COMPACT)***
  - ***compacts periodically***

GENERATIONAL COLLECTION

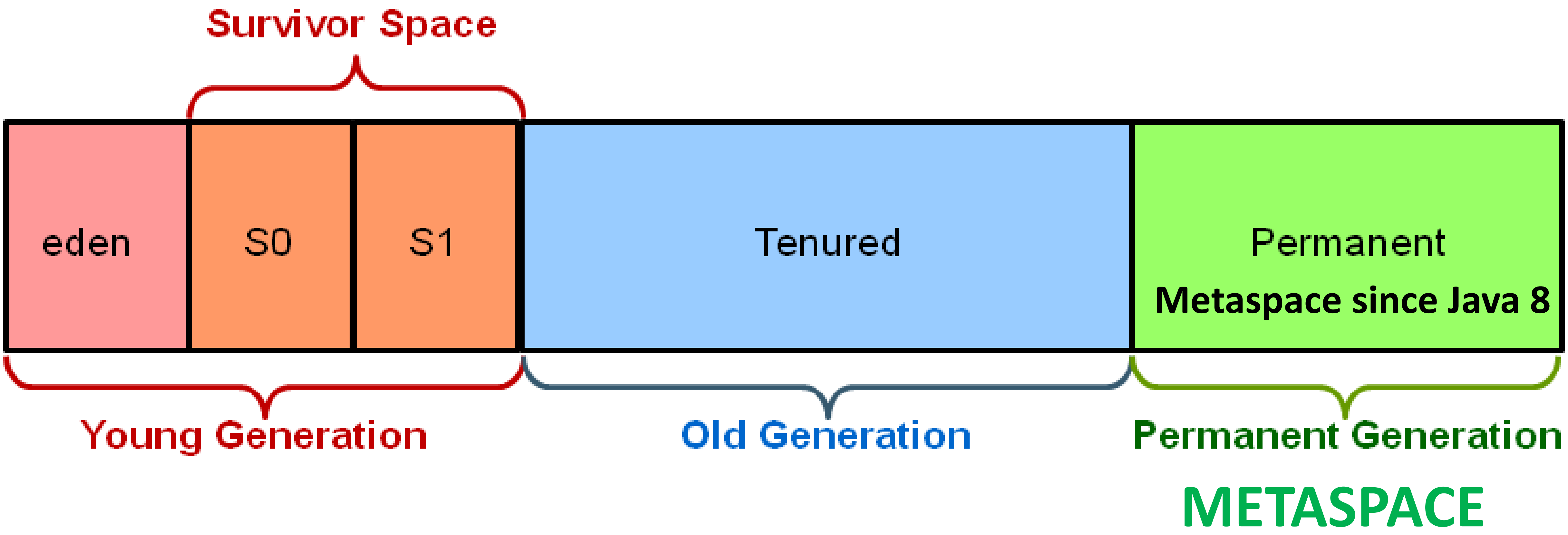
based on a 'Weak Generation' hypothesis

"MOST OBJECTS DIE YOUNG"

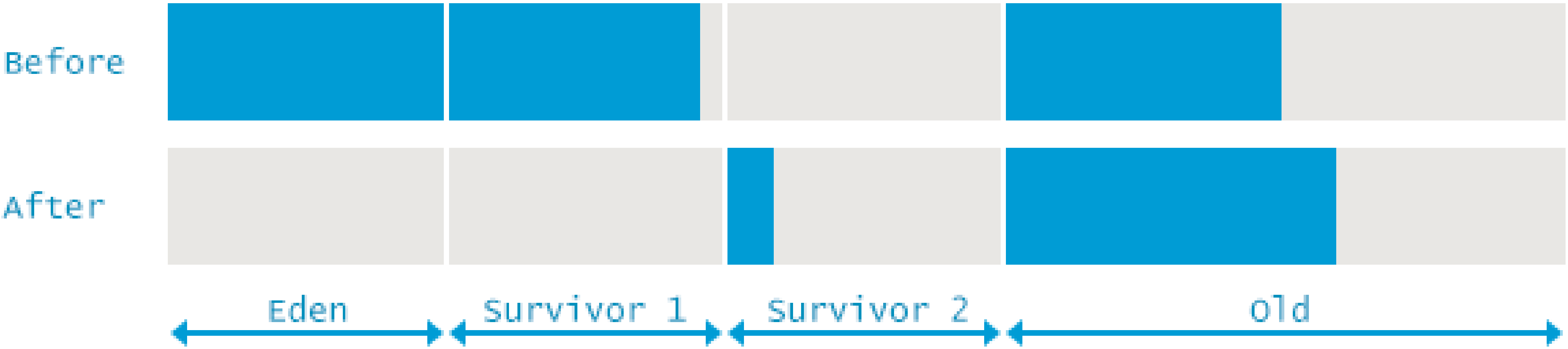
- **separate young objects from the older ones**
- **use a Copy collector (linear to the live set)**
- **promote object that are old enough to another area**
- **only collect older areas as needed**

- *requires "remembered sets"*
- *does NOT need 2x memory*

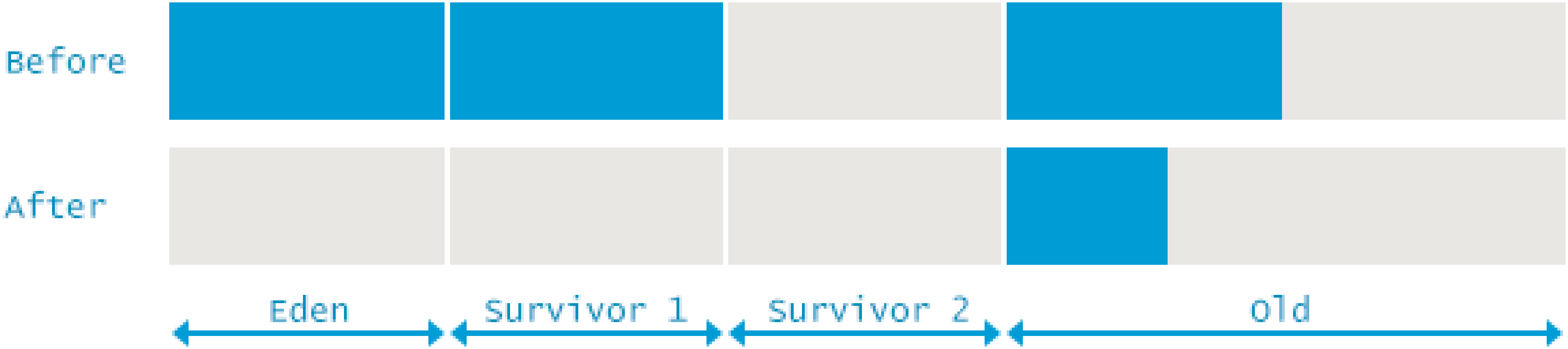
СБЕРБАНК ТЕХНОЛОГИИ



Minor GC



Full GC



СБЕРБАНК ТЕХНОЛОГИИ

Serial GC

`-XX:+UseSerialGC`

- **monolithic stop-the-world Copy in Young Generation**
- **monolithic stop-the-world Mark-Sweep-Compact in Old Generation**

Best for ~100MB heaps with single-core CPU

Parallel GC

default in Java 7 & 8

`-XX:+UseParallelGC -XX:+UseParallelOldGC`

- **monolithic stop-the-world Copy in Young Generation**
- **monolithic stop-the-world Mark-Sweep-Compact in Old Generation**

Best for multi-core CPU --> high throughput

“Mostly Concurrent Mark and Sweep Garbage Collector”

`-XX:+UseConcMarkSweepGC`

- **monolithic stop-the-world Copy in Young Generation in parallel**
- ***mostly*** **concurrent Mark-Sweep in Old Generation**

Best for *low latency*, uses application resources.

Fallback for Full Collection (monolithic STW)
when compaction is needed

СБЕРБАНК ТЕХНОЛОГИИ

G1 – "Garbage First"

default since Java 9

-XX:+UseG1GC

- monolithic stop-the-world **Copy** in Young Generation in parallel
- ***mostly*** concurrent **Mark** in Old Generation
- ***mostly*** incremental stop-the-world **Compact** in Old Generation

Best for specific goals

Tries to delay Full GC