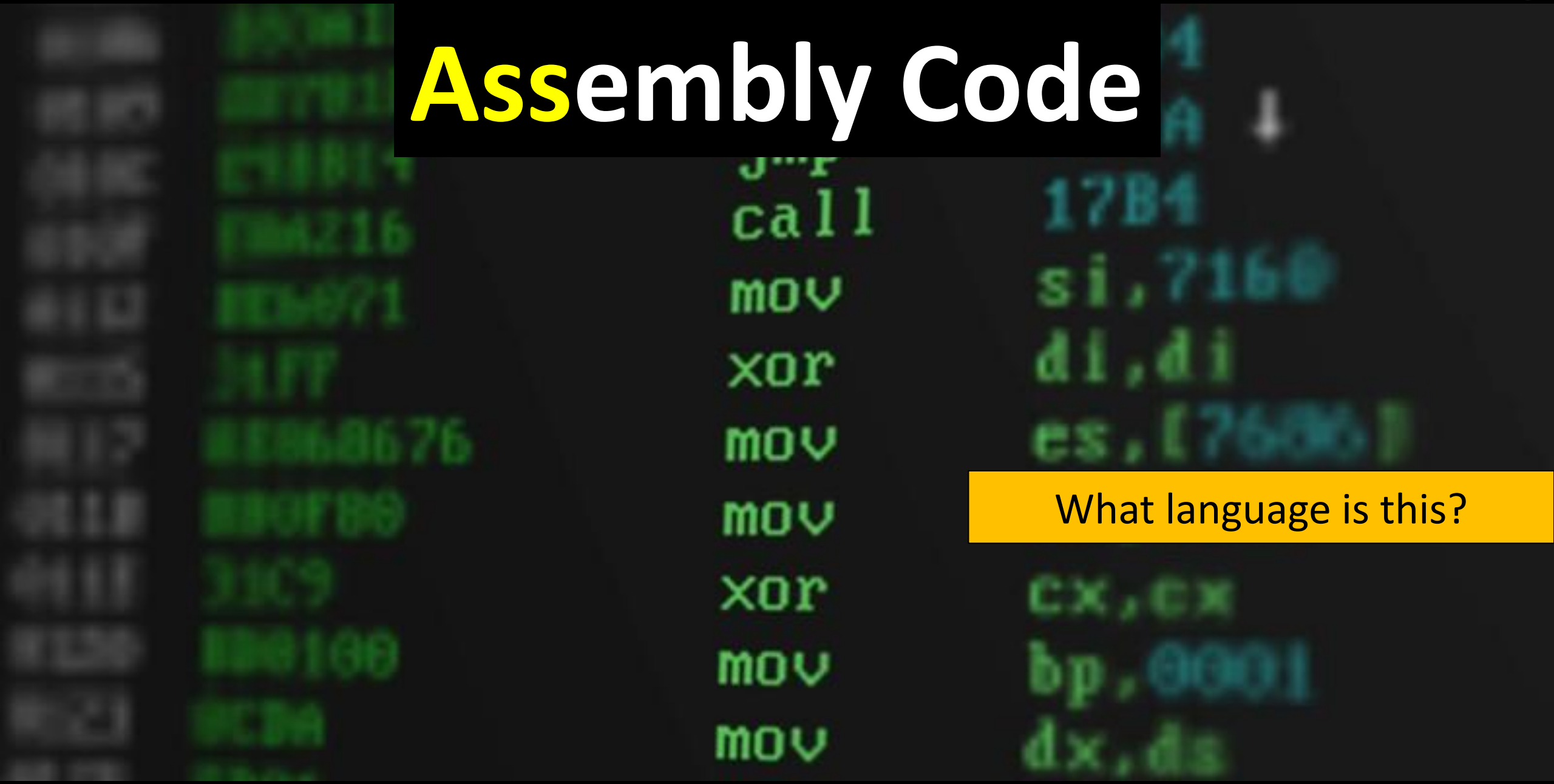
JIT

Just-In-Time Compiler

Assembly Code



What language is this?

JavaScript

code is **interpreted** by an engine (eg the browser)

✅ **running on any OS/CPU**

-- fun fact: JavaScript was created in just 10 days in 1995 by Brendan Eich!! link

```
function search() {
  const name = document.getElementById('name').value;
  fetch(`/api/search?name=${name}`)
    .then((response) ⇒ response.json())
    .then((data) ⇒ {
      const s = data
      .map(e ⇒ `<li> ${e.name} from ${e.campus}</li>`)
      .join("");
      document.getElementById('results')
            .innerHTML = `<ul>${s}</ul>`;
    });
}
```

What language is this?

C/C++

code is **compiled** by a compiler like gcc -> .exe, .o

✅ **runs faster**

```
#include<time.h>
#include<stdio.h>
int getthevalue();
void drawRectangle(int a, int b , int c, int d)
{
    rectangle(a,b,c,d);
}
void main()
{
    clrscr();
    int i,j,a,x,y;

    int interData[4][4];
    randomize();
    for(i=0;i<4;i++)
    for(j=0;j<4;j++)
    interData[i][j]=random(16);
    x=3,y=3;
    int gd=DETECT,gm;
    initgraph(&gd,&gm,"C:\TC\bgi");
```

What language is this?

is

Java

Compiled or Interpreted ?

by **javac** 👈 **BOTH** 👉 by **JVM**

```
L5
 LINENUMBER 6 L5
 IINC 3 1
 GOTO L2
L3
 LINENUMBER 9 L3
FRAME CHOP 1
 ALOAD 2
 ICONST_1
 INVOKEVIRTUAL java/lang/String.substring (I)Ljava/lang/String;
 ARETURN
L6
 LOCALVARIABLE i I L2 L3 3
 LOCALVARIABLE this Lvictor/perf/StringBuilderPlay; L0 L6 0
 LOCALVARIABLE n I L0 L6 1
 LOCALVARIABLE s Ljava/lang/String; L1 L6 2
 MAXSTACK = 2
 MAXLOCALS = 4
```

Java Bytecode

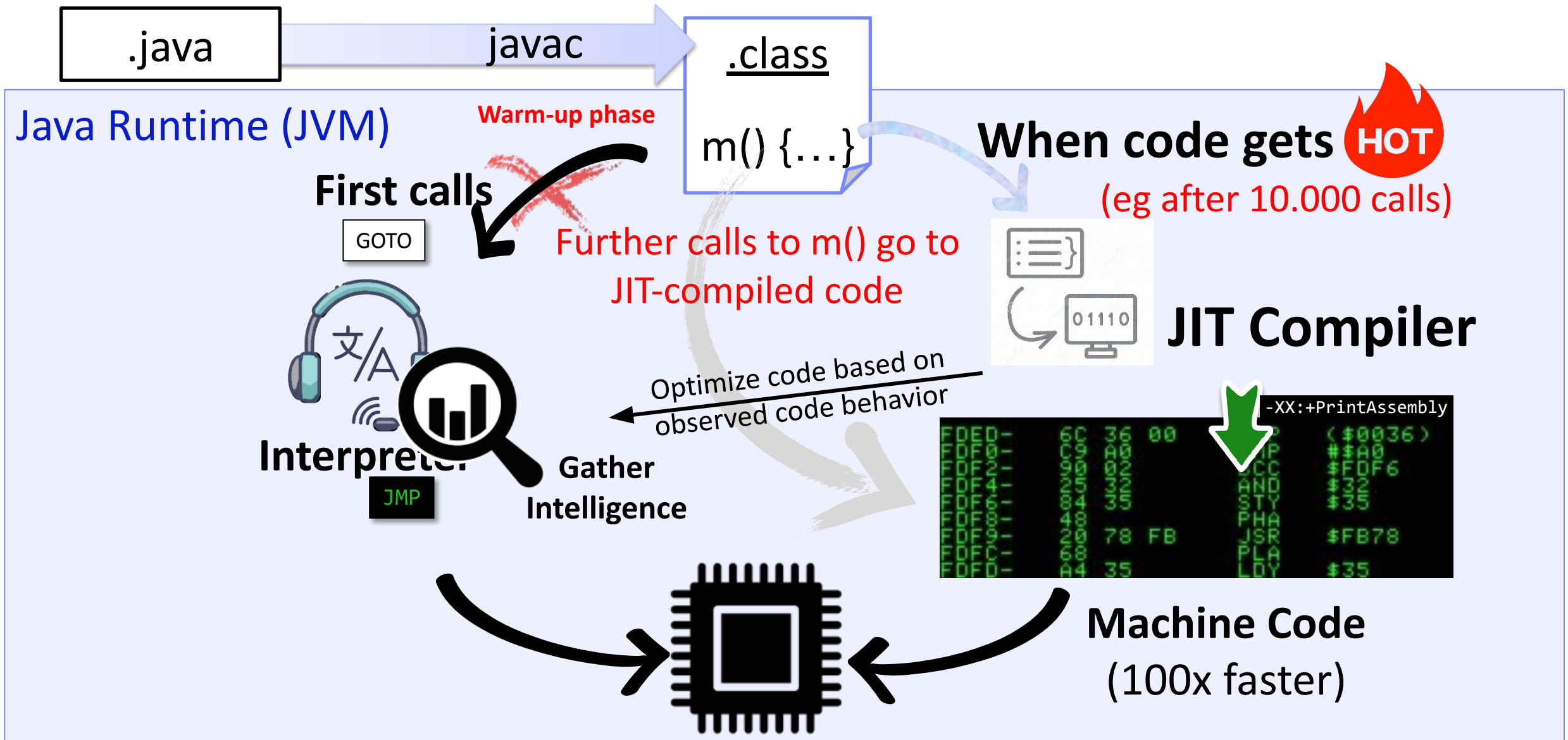
(the output of `javap My.class`)

= Idealized Machine Language (agnostic to ARM32/64, x86, AMD)

"Write Once, Run Anywhere"

What language is this?

Just In Time (JIT) Compilation



GraalVM nativeimage

- GraalVM = Java VM entirely written in Java
- Can compile .java directly to machine code = **native image**
  - As an .exe, .so, ... without any .class
- Startup time of < 1 second 😍 (vs SpringBoot/jar for 20-60s)
  - Compilation of the nativeimage can take **30-60 minutes!!!**
  - Used by Quarkus, spring-native, Cloud Functions
- Because there is no JVM:
  - No JIT optimization (no speed improvement over time)
  - Cannot use reflection or class proxies (CGLIB) □ require Ahead Of Time (AOT)

Static Optimizations

Java Source Code ⬇️ javac,gcc... optimization

```
int f(int n, int a, int b) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i * (a + b);
  }
  return sum;
}
```

```
int f(int n, int a, int b) {
  int sum = 0;
  int temp = a + b; // loop invariant
  for (int i = 0; i < n; i++) {
    sum += i * temp;
  }
  return sum;
}
```

Don't micro-optimize: let JIT do it!

Speculative Optimizations

called 10.000 times with **debug** = **false**

```
  if (debug) {
    log.debug("Debug: " + x);
  } else {
    someBusinessLogic();
  }
}
```

Very low performance loss

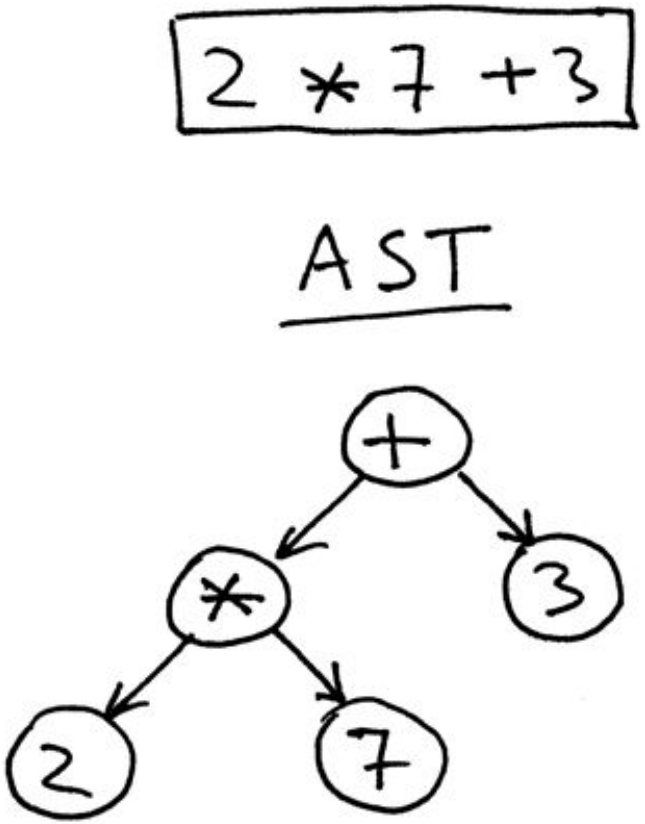
debug becomes **true** later

```
if (!debug) {
  someBusinessLogic();
} else {
  trap();
}
```

Deoptimization

revert to interpreter

Abstract Syntax Tree (AST)



```
if n|3: fizz
if n|3: buzz
else n

Output:

1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizz buzz
```

Suppose this is called 10K times 🔥

```java
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };
    FizzFuzz fuzz = new FizzFuzz(1, 100);
    fuzz.printNumbersDivisibleByN(divisors, messages);
  }

  private int lowerRange;
  private int upperRange;

  public FizzFuzz(int lowerRange, int upperRange) {
    this.lowerRange = lowerRange;
    this.upperRange = upperRange;
  }

  public int getLowerRange() {return this.lowerRange;}
  public int getUpperRange() {return this.upperRange;}

  public void printNumbersDivisibleByN(int[] n, String[] messages) {
    if ((n == null) || (messages == null)) return;
    for (int i = getLowerRange(); i < getUpperRange(); i++) {
      String messageToPrint = fizzFuzz(i, n, messages);
      System.out.println(messageToPrint);
    }
  }

  private String fizzFuzz(int value, int[] div, String[] messages) {
    if ((div == null) || (messages == null)) return "";
    int index = div.length;
    while (--index >= 0)
      if ((value % div [index]) == 0)
        return messages[index];
    return Integer.toString(value);
  }
}
```

JIT Optimizations

- Inline accessors (getters+setters)

they don't exist at runtime

unless you set a breakpoint in them

Fizz-Fuzz Exercise

Print numbers from 1 to 100.

But for multiples of three print “fizz”.

For the multiples of five print “buzz”.

For numbers which are multiples of both three and five print “fizz buzz“.

Some Optimizations

- Inline accessors

Suppose this is called 10K times

```
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };
    FizzFuzz fuzz = new FizzFuzz(1, 100);
    fuzz.printNumbersDivisibleByN(divisors, messages);
  }

  private int lowerRange;
  private int upperRange;

  public FizzFuzz(int lowerRange, int upperRange) {
    this.lowerRange = lowerRange;
    this.upperRange = upperRange;
  }

  public void printNumbersDivisibleByN(int[] n, String[] messages) {
    if ((n == null) || (messages == null)) return;
    for (int i = lowerRange; i < upperRange; i++) {
      String messageToPrint = fizzFuzz(i, n, messages);
      System.out.println(messageToPrint);
    }
  }

  private String fizzFuzz(int value, int[] div, String[] messages) {
    if ((div == null) || (messages == null)) return "";
    int index = div.length;
    while (--index >= 0)
      if ((value % div [index]) == 0)
        return messages[index];
    return Integer.toString(value);
  }
}
```

```java
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };
    FizzFuzz fuzz = new FizzFuzz(1, 100);
    fuzz.printNumbersDivisibleByN(divisors, messages);
  }

  private int lowerRange;
  private int upperRange;

  public FizzFuzz(int lowerRange, int upperRange) {
    this.lowerRange = lowerRange;
    this.upperRange = upperRange;
  }




  public void printNumbersDivisibleByN(int[] n, String[] messages) {
    if ((n == null) || (messages == null)) return;
    for (int i = lowerRange; i < upperRange; i++) {
      String temp = Integer.toString(i);
      if ((n == null) || (messages == null)) temp = "";
      int index = n.length;
      while (--index >= 0)
        if ((i % n[index]) == 0) {
          temp = messages[index];
          break;
        }
      System.out.println(temp);
    }
  }
}
```

Suppose this is called 10K times

Some Optimizations

- Inline accessors
- Inline to broaden horizon

Only for reasonably small methods

```
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };
    FizzFuzz fuzz = new FizzFuzz(1, 100);
    fuzz.printNumbersDivisibleByN(divisors, messages);
  }

  private int lowerRange;
  private int upperRange;

  public FizzFuzz(int lowerRange, int upperRange) {
    this.lowerRange = lowerRange;
    this.upperRange = upperRange;
  }




  public void printNumbersDivisibleByN(int[] n, String[] messages) {
    if ((n == null) || (messages == null)) return;
    for (int i = lowerRange; i < upperRange; i++) {
      String temp = Integer.toString(i);
      int index = n.length;
      while (--index >= 0)
        if ((i % n[index]) == 0) {
          temp = messages[index];
          break;
        }
      System.out.println(temp);
    }
  }
}
```

Some Optimizations

- Inline accessors
- Inline to broaden horizon
- Loop invariant

```java
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };

    int lowerRange = 1;
    int upperRange = 100; //on caller's stack instead of heap

    if ((n == null) || (messages == null)) return;
    for (int i = lowerRange; i < upperRange; i++) {
      String temp = Integer.toString(i);
      int index = divisors.length;
      while (--index >= 0)
        if ((i % divisors[index]) == 0) {
          temp = messages[index];
          break;
        }
      System.out.println(temp);
    }
  }
}
```

Some Optimizations

- Inline accessors
- Inline to broaden horizon
- Loop invariant
- Inline to caller
  - Escape analysis
- Constant propagation

```java
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };

    if ((divisors == null) || (messages == null)) return;
    for (int i = 1; i < 100; i++) {
      String temp = Integer.toString(i);
      int index = divisors n.length;
      while (--index >= 0)
        if ((i % divisors[index]) == 0) {
          temp = messages[index];
          break;
        }
      System.out.println(temp);
    }
  }
}
```

Some Optimizations

- Inline accessors
- Inline to broaden horizon
- Loop invariant
- Inline to caller
  - Escape analysis
- Constant propagation

Some Optimizations

```
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };

    for (int i = 1; i < 100; i++) {
      String temp = Integer.toString(i);
      int index = 3;
      while (--index >= 0)
        if ((i % divisors[index]) == 0) {
          temp = messages[index];
          break;
        }
      System.out.println(temp);
    }
  }
}
```

- Inline accessors
- Inline to broaden horizon
- Loop invariant
- Inline to caller
  - Escape analysis
- Constant propagation
- Loop Unrolling

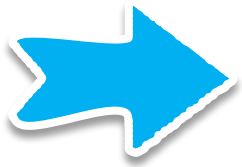
```
for (int i = 0; i < 100; i++) {
   method(i);
} // 100 JMPs back up
```

```
for (int i = 0; i < 100; i += 5) {
   method(i);
   method(i+1);
   method(i+2);
   method(i+3);
   method(i+4);
} // 20 JMPs ✅
```

https://blogs.oracle.com/javamagazine/loop-unrolling

Article: Enhanced for is also unrolled

```
for (Order order : orders) {
   // a bit of code
}
```



Split Loop Refactor

Split a large loop into many small ones
may work faster!!

```java
public class FizzFuzz {
  public static void f() {
    int[] divisors = { 3, 5, 15 };
    String[] messages = { "fizz", "buzz", "fizz buzz" };

    for (int i = 1; i < 100; i++) {
      String temp = Integer.toString(i);
      int index = 3;
      if ((i % divisors[--index]) == 0)
        temp = messages[index];
      else if ((i % divisors[--index]) == 0)
        temp = messages[index];
      else if ((i % divisors[ --index]) == 0)
        temp = messages[index];
      System.out.println(temp);
    }
  }
}
```

messages[2] -> "fizz fuzz"

```java
while (--index >= 0)
  if ((i % divisors[index]) == 0) {
    temp = messages[index];
    break;
  }
```

Some Optimizations

- Inline accessors
- Inline to broaden horizon
- Loop invariant
- Inline to caller
  - Escape analysis
- Constant propagation
- Loop Unrolling
- Constant Propagation

```
if n|3: fizz
if n|3: buzz
else n

Output:

1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizz buzz
```

```java
public class FizzFuzz {
  public static void f() {
    for (int i = 1; i < 100; i++) {
      String temp = Integer.toString(i);
      if (i % 15 == 0)
        temp = "fizz buzz";
      else if (i % 5 == 0)
        temp = "fuzz";
      else if (i % 3 == 0)
        temp = "bizz";
      System.out.println(temp);
    }
  }
}
```

Works as fast as the first version
(Proved with JMH Benchmarks)

Moral ?

```java
public class FizzFu
  public static voi
    int[] divisors
    String[] messag
    FizzFuzz fuzz =
    fuzz.printNumbe
  }

  int lowerRange, u

  public FizzFuzz(i
    this.lowerRange
    this.upperRange
  }

  public int getLow
  public int getUpp

  public void print
    if ((n == null)
    for (int i = ge
      String messag
      System.out.pr
    }
  }

  private String fi
    if ((div == nul
    int index = div
    while (--index
      if ((value %
        return mess
    return Integer.
  }
}
```

Some Optimizations

- Inline accessors
- Inline to broaden horizon
- Loop invariant
- Inline to caller
  - Escape analysis
- Constant propagation
- Loop Unrolling
- Constant Propagation

Moral ?

- **Don't micro-optimize!**
  - JIT micro-optimizes it anyway!
  - Keep your code clean - JIT optimizes better small & encapsulated code
- **Don't Guess, Measure:**
  - On production machines + load -- or load tests
  - Compare measured gain ⏱️ vs added code complexity & risk.
  - A classic: ~~"parallelStream should work faster"; StringBuilder.append(...)~~
- **Java starts slow!**
  - JIT optimizations accelerates Java execution
  - In benchmarks, ignore the performance of the first warmup iterations.

Keep Your Code Clean

If a flow is CPU-bound (doing no/little I/O) and you tried everything else (profiling..)

Is my code JIT-Friendly ?

Warning: this makes sense for high-frequency endpoints with response times ≅ milliseconds

1. Identify methods on the hot paths to optimize

2. Check that these methods do NOT appear in the jarscan output
   (= means they are inline-able)

3. Check that the methods appear in the -XX:+PrintCompilation prod
   (= means that they got warm enough to be compiled by JIT)

https://www.infoq.com/articles/Java-Application-Hostile-to-JIT-Compilation/

How .java turns into ASM:

📺 From **Java to Assembly** - Down the rabbit hole

by Charles Nutter

https://www.youtube.com/watch?v=HBKVdJph_oQ

Monitoring and Tuning JIT

Think of this in projs with HUGE codebases and frameworks

Code Cache

`-XX:+PrintCodeCache`

- Stores machine code compiled by JIT
- Too small?
  - JIT Compilation stops
  - **Huge Performance penalty !**
  - Solution: max *= 2 or 4 ☺

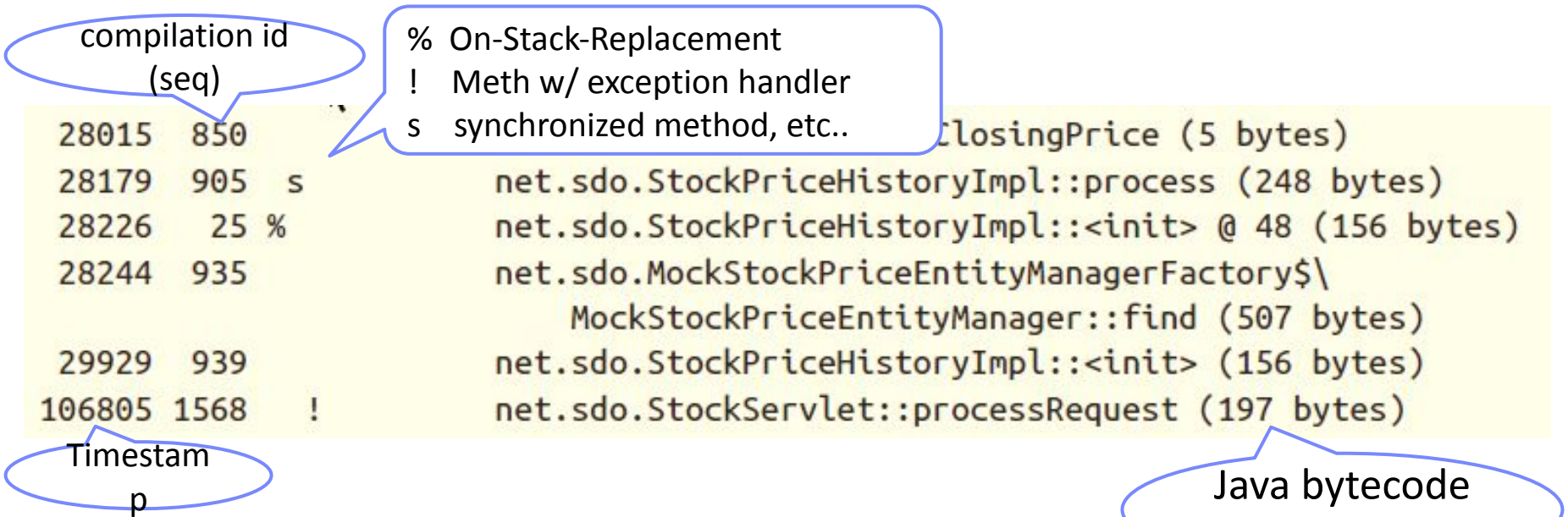
You should never see this after adding -XX:+PrintCodeCache to jvm options:

```
Java HotSpot(TM) 64-Bit Server VM warning: CodeCache is full.
         Compiler has been disabled.
Java HotSpot(TM) 64-Bit Server VM warning: Try increasing the
         code cache size using -XX:ReservedCodeCacheSize=
```

-XX:CompileThreshold=*N*

- Lower it if your critical path is not JIT compiled
- Method counters cool back down
  - A **lukewarm method** is invoked often, but never **HOT enough** for JIT
  - Alternative: stop cooling down with -XX:-UseCounterDecay

-XX:+PrintCompilation



jstat -compiler <pid>

```
Compiled Failed Invalid   Time   FailedType FailedMethod
     206      0       0     1.97          0
```

jstat -printcompilation <pid> <rate>

```
Compiled  Size  Type Method
     207    64     1 java/lang/CharacterDataLatin1 toUpperCase
     208     5     1 java/math/BigDecimal$StringBuilderHelper getCharArray
```

Deoptimization

- **Not entrant**: Code needs to be recompiled
  - Due to broken assumptions
  - Or Tiered Compilation: code is compiled with a better compiler
- **Zombie**
  - All the instances using that compiled version are gone

```
 841113   25 %          net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                               made not entrant
 841113  937  s          net.sdo.StockPriceHistoryImpl::process (248 bytes)
                               made not entrant
1322722   25 %          net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                               made zombie
1322722  937  s          net.sdo.StockPriceHistoryImpl::process (248 bytes)
                               made zombie
```

Tiered Compilation log

- 0: Interpreted code
- 1: Simple C1 compiled code
- 2: Limited C1 compiled code
- 3: Full C1 compiled code
- 4: C2 compiled code

Constructor goes from level 3 to level 4 compilation:

```
40915   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
40923 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41418   87 %     4       net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
41434   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                 made not entrant
41458 3749       4       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41469 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                                 made not entrant
42772 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                                 made zombie
42861   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                 made zombie
```

Method Inlining

-XX:+PrintInlining

- **Small Hot** methods are automatically inlined
  - Example: getters/setters

```
Point p = getPoint();
p.setX(p.getX() * 2);
```

```
public class Point {
    private int x, y;

    public void getX() { return x; }
    public void setX(int i) { x = i; }
}
```

  - Is compilec

```
Point p = getPoint();
p.x = p.x * 2;
```

```
.InliningExample::calculateSum (12 bytes)   hot method too big
```

  - Never-ever do this: -XX:-Inline
  - Large methods are not inline -> jvm flag

Hotspot Monitoring

- Options to check on code
  - -XX:+PrintCompilation
  - -XX:+PrintSignatureHandlers
  - -XX:+LogCompilation
    - requires -XX:+UnlockDiagnosticVMOptions
  - -XX:+PrintInlining
  - -XX:+PrintNMethods
  - prints methods that are being JITed
- Possible to dump assembler
  - requires a fast debug build of the JVM
  - -XX:+PrintAssembly

$O(N)$

Complexities and Data Structures

Big-O notation

$O(f)$ = The set of functions that grow asymptotic slower or equal to f

Examples:

- O(1) – constant time, but how large constant?
  - 20ms, 1 sec, 1min ? Example: sort by 1 external WS call
- O(logN) – binary search or trees (TreeMap/Set)
- O(N) – one sweep of the data
- O(N logN) – quick-sort, fastest comparison-based sorting
- $O(N^2)$ – bubble-sort (avoid!)

Using Big-O notation

- Reason about Time
  - Focus on the ops taking longest time
  - The constant factor is critical: c x $f(N)$
    - = how heavy is one operation
- Reason abut Memory
  - Let's store SHA(input) = O(1) not raw input = O(N)
- Calculus:
  - $O(N^2) > O(N \log N) > O(N) > O(\log N)$
  - $O(N) + O(N \log N) = O(N \log N)$
  - $O(N) + O(N) = O(N)$

```
for i = 1..N
    for j = 1..N
        sum +=i*j;
for i = 1..N
    HTTPcall(i)
```

ArrayList<String> **vs** String[]

which occupies more memory for 50 elements?

Why?

Collection methods with O(n) performance should be used carefully

Intentionality | Not efficient Maintainability java:S2250

Why is this an issue?

The time complexity of method calls on collections is not always obvious. For instance, for most collections the `size()` method takes constant time, but the time required to execute `ConcurrentLinkedQueue.size()` is O(n), i.e. directly proportional to the number of elements in the collection. When the collection is large, this could therefore be an expensive operation.

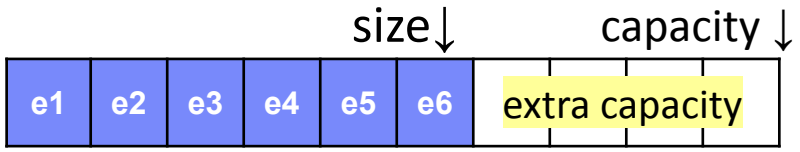
This rule raises an issue when the following O(n) methods are called outside of constructors on class fields:

- `ArrayList`
  - `contains`
  - `remove`
- `LinkedList`
  - `get`
  - `contains`
- `ConcurrentLinkedQueue`
  - `size`
  - `contains`
- `ConcurrentLinkedDeque`
  - `size`
  - `contains`
- `CopyOnWriteArrayList`
  - `add`
  - `contains`
  - `remove`
- `CopyOnWriteArraySet`
  - `add`
  - `contains`
  - `remove`

Noncompliant code example

```
ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
//...
log.info("Queue contains " + queue.size() + " elements");  // Noncompliant
```

ArrayList



Array int[]

`get(index)` = *O(1)* = direct memory access: `int v = *(arr + index);`

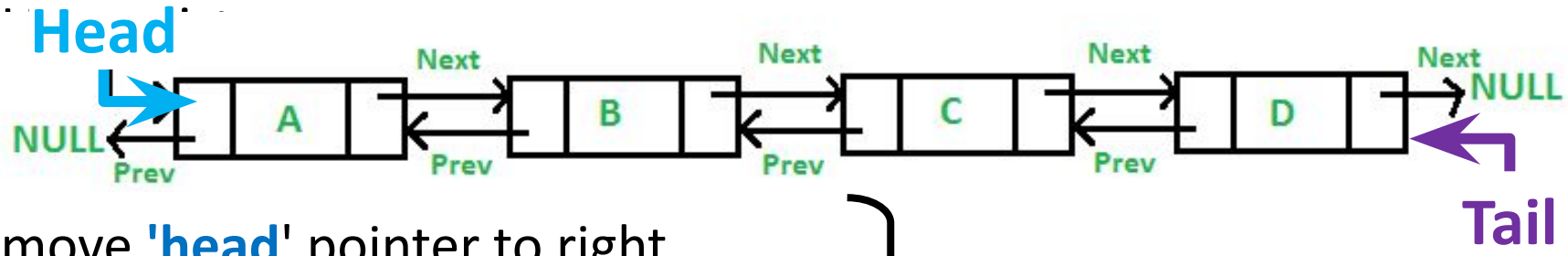
`add(e)` = *O(1) (amortized)* arr[size++] ⚠️when exceeds capacity (rare), reallocates a larger contiguous array + 50% its capacity, and copies all the data = O(N)

`stream.filter(->).findFirst()` = *O(N)* in the worst case you must visit all elements

`contains(x)` = *O(N)* .equals(x) on each element ➡️ use Set<>

`remove(0)` = *O(N)* all elements shift to the left; better ⬇️ use

LinkedList



`remove(0)` = *O(1)* move **'head'** pointer to right

`add(e)` = *O(1)* add to end and move **'tail'** pointer

= Best for a Queue

`get(i)` = *O(N)* requires N "jumps" on "Next" ref

+

extra-memory-heavy ⚠️

Set<>

The Set is actually a Map!

HashSet<E> stores elements as keys in a HashMap<E,>

```
public class HashSet<E> ... {
  private static final Object PRESENT = new Object();
  private transient HashMap<E,Object> map;
  public boolean add(E e) { return map.put(e, PRESENT)==null; }
  public boolean remove(Object o) { return map.remove(o)==PRESENT; }
  public boolean contains(Object o) { return map.containsKey(o); }
  ...
}
```

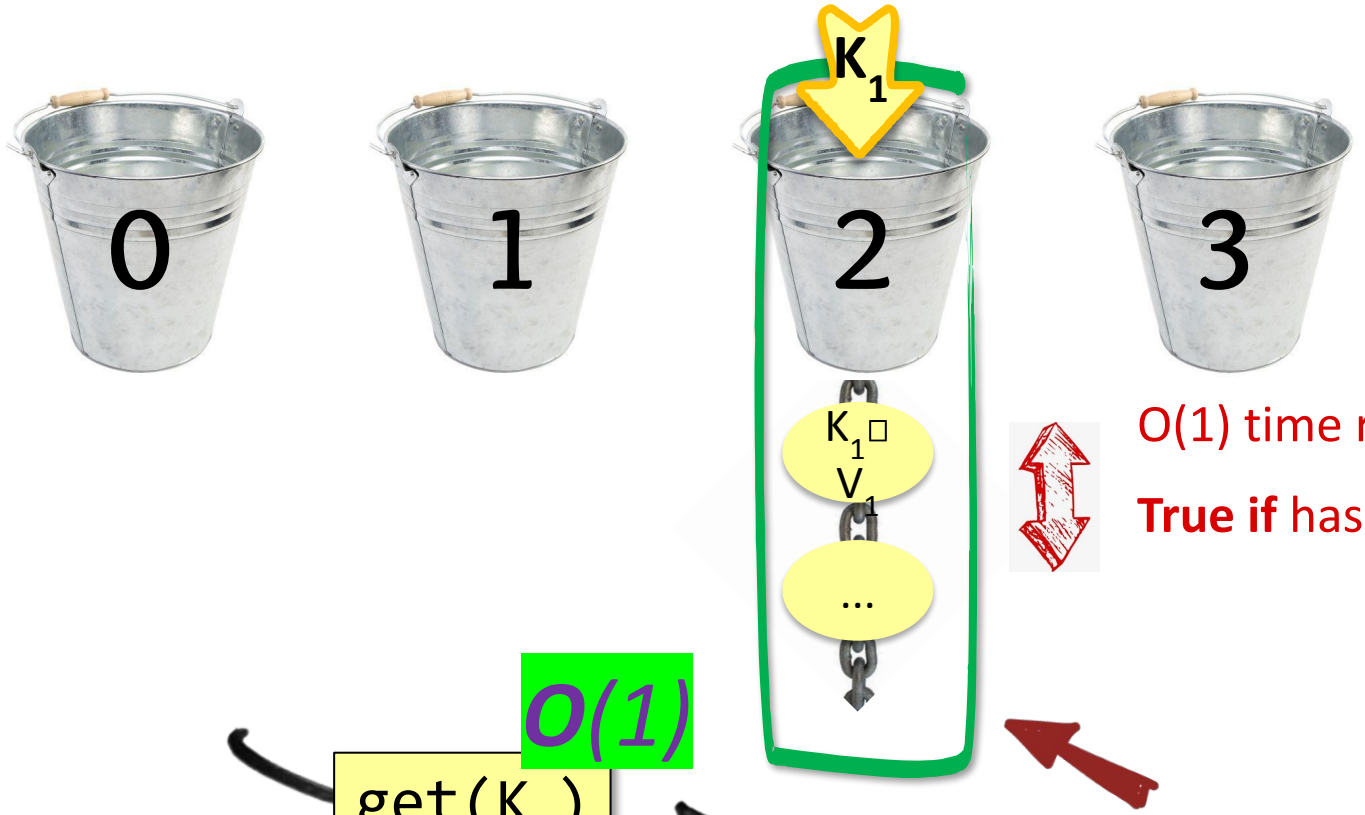
HashMap

(aka hashtable)

O(1)

put(K_1,V_1)

hash(K_1) = 10 % 4 = 2



When many elements accumulate, HashMap doubles its buckets and redistributes all entries

O(1) time requires bucket size to be ~constant.

True if hashCode() **distributes keys uniformly** in buckets

O(1)

get(K_2)

containsKey(K_2)
remove(K_2)

hash(K_2) = 10 % 4 = 2

How to break HashMap performance:

keys with hashCode() *eg* {**return** 1;}

HashMap degrades to a LinkedList O(N), not ~~O(1)~~

Since Java8, entries in a bucket are kept in a balanced tree (logN speed)

DON'T! Instead: use Objects.hash(a,b,c...)