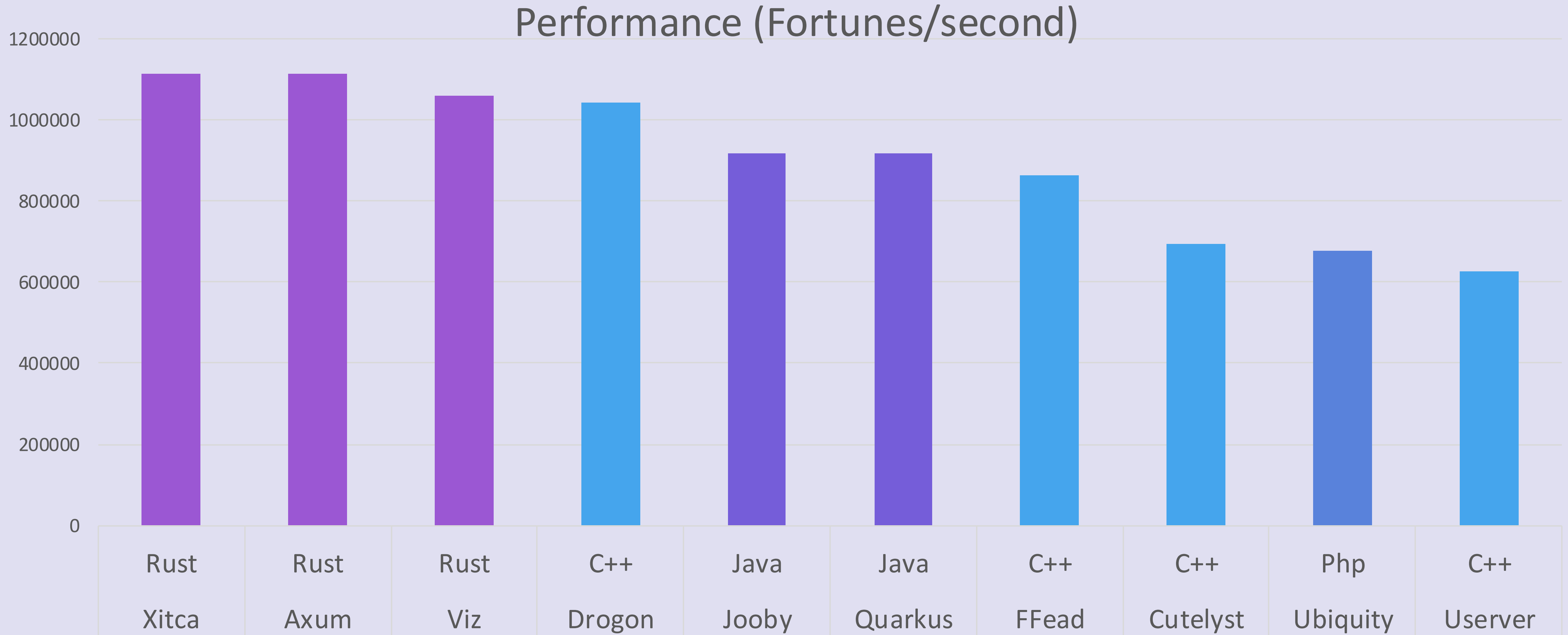
The Java Virtual Machine

Java is fast, actually?



Source: TechEmpower

How Programming Languages Execute

- Interpreted Languages
  - Python, JavaScript**, Ruby, PHP, Perl
- Compiled Languages
  - Rust, Swift, Go, C/C++

- Java is not entirely compiled nor interpreted
- JIT (Just-in-time) Compiler – compile to bytecode, interpreted at runtime
  - Worst of both worlds?

Java Virtual Machine (JVM)

- JVM translates .java files to bytecode, an intermediary representation (IR) of machine code

```
ıer.java
public class Adder {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

```
Compiled from "Adder.java"
public class Adder {
  public Adder();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static int add(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: ireturn
}
```

Bytecode (IR) vs Machine Code

- **Bytecode is translated to machine code Just-in-time (JIT)!**

```
Compiled from "Adder.java"
public class Adder {
  public Adder();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>"
       4: return

  public static int add(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: ireturn
}
```

```
add:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-8]
        add     eax, edx
        pop     rbp
        ret
```

Why use JIT

- Bytecode can be type-checked at compile time (nearly no runtime checks => fast and safe)
- Good compilers aggressively optimize your code to make it faster
- At compile-time, the compiler does not know about what parts of your code run often (method you use 1x, vs 1000x, or if a method simplifies into something optimizable)
- At run-time, the JVM knows more about what parts of your code run frequently
- Can make more intelligent optimizations with runtime context

Bytecode Verification

- What if someone writes malicious .class files?
- Static verification of type, memory, control flow, stack safety to ensure no malicious or malformed bytecode prior to execution
- Structural checks
  - Magic number 0xCAFEBABE
  - Constant pool integrity
  - Valid method and field descriptors
- Semantic checks/symbol verification
  - Referenced classes/methods exist
  - Access control rules are respected

Bytecode Verification

- Type and Stack verification
- Simulates execution using dataflow analysis
- Tracks the types of values on the operand stack and local variables
- Every operation must have correct type of operands
- Branch targets must lead to instructions with compatible stack states
- No stack overflows or underflows

Why Bytecode Verification?

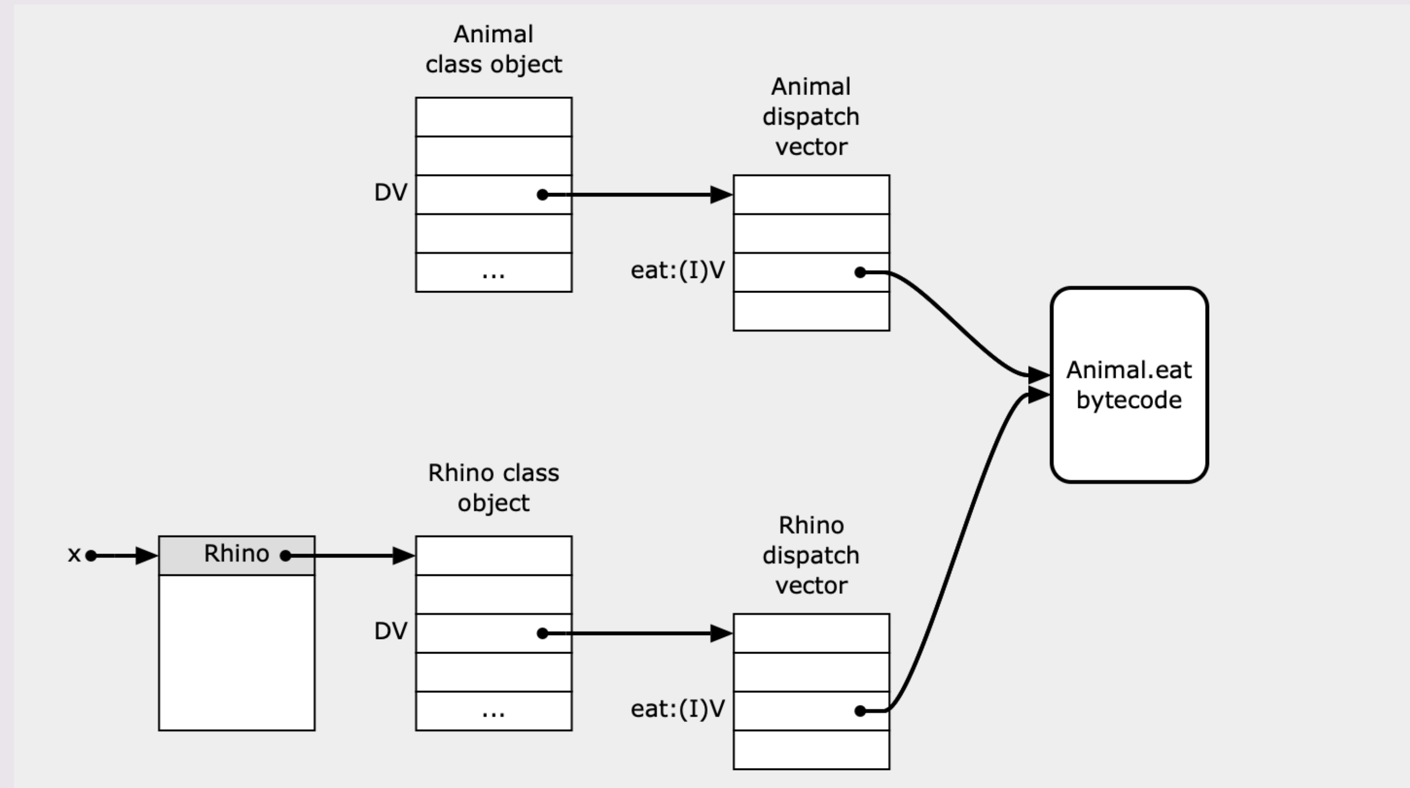
- Ensures sandboxing (no malicious bytecode injection)
- Verified type and memory safety
- No segmentation faults, ever (huge problem in C, C++)
  - This is SUPER important!
- Enforces, provably, security and correctness for all bytecode programs prior to execution
- Enhances both speed and security during execution

Method Dispatch

- How should the JVM find which overloaded method to call?
- It could resolve overloading at run-time: perform a run-time check on the signature of the method
- It could resolve overloading at compile-time, and substitute overloaded calls for new, non-overloaded calls
- Java does the latter for speed and optimization reasons: “method dispatch”
- Allows for polymorphism with limited overhead at runtime

Method Dispatch

- Each object points to its class object (e.g., there is some “String” object)
- Contains a **dispatch table:** an array of method entry pointers
- When `invokevirtual` is called, Java looks up method call in the dispatch table
- Jumps to the method specified in the table
- Subclasses share the inherited slots; overwrites overridden methods
- An invocation of such a method is called a **call site**



Method Dispatch

Instruction	What it Calls	Dispatch Needed	Example
invokestatic	Static methods	No	Math.max()
invokespecial	Private/constructors/final	No	super()
invokevirtual	Normal Instance Methods	Yes	cat.meow();
invokeinterface	Interface Methods	Yes (slow!)	pet.adopt();
invokedynamic	Dynamic Call Sites	Runtime linkage (slow)	Reflection, lambdas

Stage 1: Interpretation

- Maps each bytecode instruction to machine code (platform dependent)
- Template interpreter – direct pre-compiled machine code rather than pure interpretation (with caveat: dispatch tables)
- Extremely fast startup and memory efficient
- Slow execution (significant overhead per bytecode instruction)
- Good for code that runs once or twice or infrequently (most code)

Stage 2: Profiling

- JVM profiles program during execution for optimization opportunities
  - Method invocation counters (how many times method is called)
  - Loop iteration counters
  - Branch profiling
  - Type profiling
  - Exception frequency
- At 1,500 invocations, queued for C1 (client) compiler
- At 10,000 invocations, queued for C2 (server) compiler

Stage 3: Native Compilation

- Once method crosses threshold, compiled into optimized native machine code
- In code cache, and method dispatch updates directly to cached version
- Cache must be maintained and pruned occasionally (~240MB)
- Two different tiers of compilers: C1 (Client) and C2 (Server)

C1 Compiler

- Activated at ~1,500 invocations
- Moderate optimizations
  - Linear scan register allocation (simple)
  - Basic inlining
  - Constant folding
  - Null check elimination
  - Dead code elimination
- Compiles with profiling instrumentation for promotion to C2
- Compilation is extremely fast (milliseconds)

C2 Compiler

- Activated at ~10,000 invocations
- Slow, aggressive optimizations (100x longer than C1)
- 10-100x faster than interpreted bytecode, 2-10x faster than C1, often exceeds handwritten C++

C2 Compiler Optimizations

- Method inlining across class boundaries
- Loop optimizations (hoisting, vectorization, unrolling)
- Escape analysis, scalar replacement
- Global code motion and scheduling
- Advanced register allocation w/ graph coloring
- Sea-of-Nodes IR (SSA-based dataflow graph) for further optimizations

Inlining

- Inlining – replacing method call with method body
- Eliminates call stack overhead and enables further optimization
- Includes getters and setters – there is no performance cost!

```
public class Adder {
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int primaryResult = add(1, 2);
        int secondaryResult = add(4, 5);
        return add(primaryResult + secondaryResult);
    }
}
```

```
public class Adder {
    public static void main_inlined(String[] args) {
        // You can imagine that the compiler can distill this to a constant!
        return ((1 + 2) + (4 + 5));
    }
}
```

Loop Optimizations

```
// Regular loop
int[] array = new int[1000];
for (int i = 0; i < 1000; i++) {
    array[i] = i * 2;
}
```

```
// Unrolling (take advantage of parallelism)
// Can also use Vectorization or SMD
for (int i = 0; i < 1000; i += 4) {
    array[i] = i * 2;
    array[i + 1] = (i + 1) * 2;
    array[i + 2] = (i + 2) * 2;
    array[i + 3] = (i + 3) * 2;
}
```

```
// Strength reduction (+ is faster than *)
int temp = 0;
for(int i = 0; i < 1000; i++) {
    array[i] = temp;
    temp += 2;
}
```

Escape Analysis

- If an object is confined to a scope or thread, we can make big optimizations (especially with heap allocation or synchronization!)

```
public static int distanceFromOrigin(int x, int y) {
    Point p = new Point(x:1, y:2);
    return p.x * p.x + p.y + p.y;
}

public static int distanceFromOrigin_optimize(int x, int y) {
    // Constructor
    int p_x = x;
    int p_y = y;
    // (with no heap allocation, no "new" call!)
    return p_x * p_x + p_y * p_y;
}
```

Escape Analysis (cont.)

- Code shared across multiple threads needs to be synchronized
- In practice, that means adding a wrapper object that "locks" (additional overhead) – but if single threaded, no need

```
public static String stringBuilderDemo() {
    // This StringBuilder never leaves the thread
    StringBuilder output = new StringBuilder();
    output.append(str:"Hello, ");
    output.append(str:"World!");
    return output.toString();
    // So the JIT Compiler will not perform any synchronization on it
    // It will not lock the ovject
    // Eliminating more overhead (esp. wrt. atomicity/synchronization)
}
```

Speculative Optimizations

- Idea: if some condition enabling optimization is true, we run optimized code
  - If we can optimize something for 99% of inputs, but the possibility of an input that occurs 1% of the time is preventing it, we want to still run the optimized code on most inputs
- If that condition is false, we run unoptimized code (deoptimization) ensuring total correctness
- To enable this kind of optimization, profiling necessary to find optimizable inputs and frequencies
- Limited to C2

Miscellaneous Optimizations

- Dead code elimination (i.e. DEBUG constant set in the program)
- Constant forwarding (replace all instants of a constant with literal)
  - Sometimes, something becomes constant non-deterministically.
  - JIT can still detect this and optimize in those cases!
- Range check elimination for loops (no more comparing length of array)
- Optimizing for common types and inputs (i.e. if you use a class 90% in a polymorphic call, will optimize for that 90%)

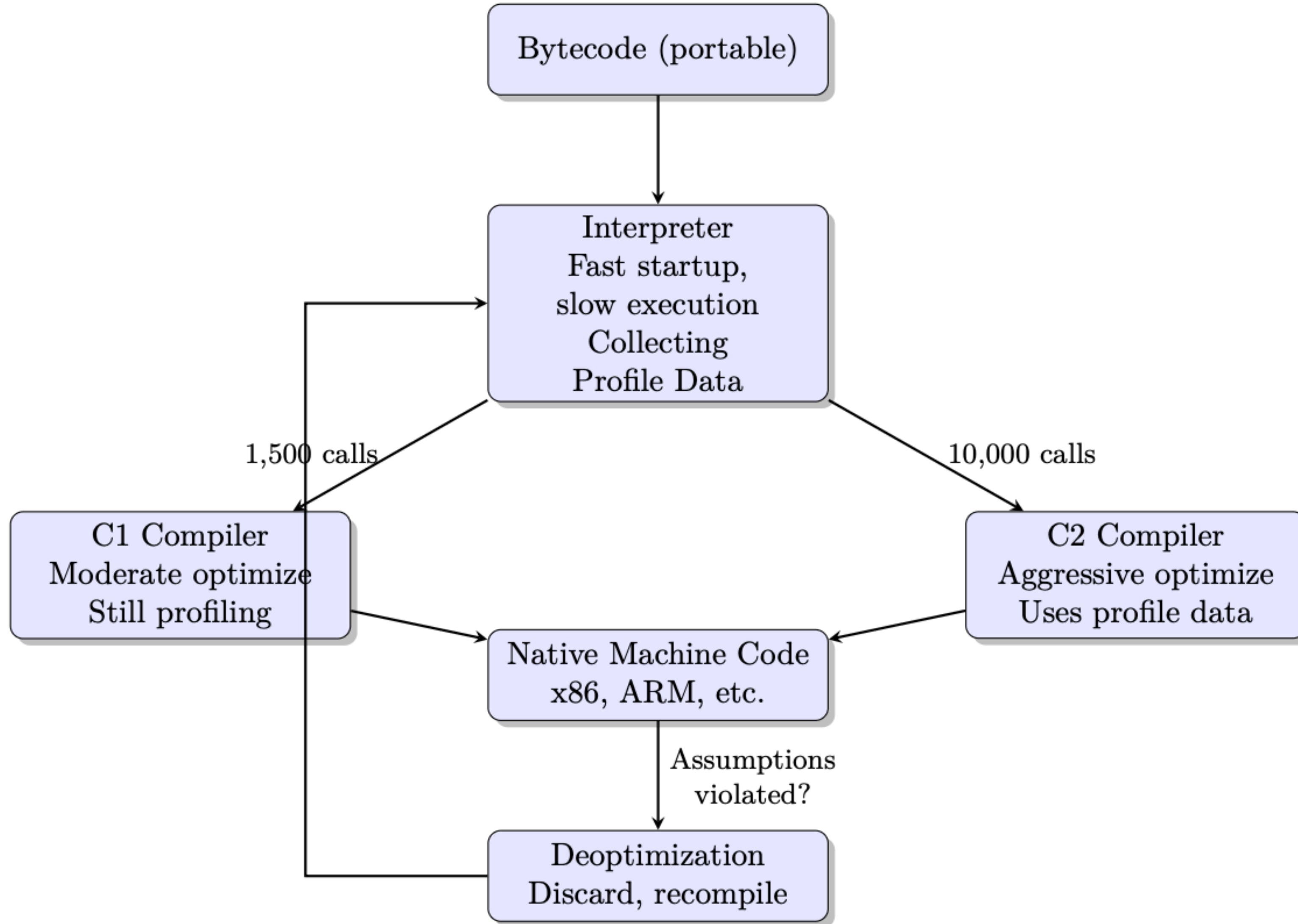


Figure 1: JVM Adaptive Execution and Feedback Loop

Coding for the JIT

- Write for clarity and abstraction first; performance second
  - The JIT is really good at optimizing abstractions!
- Final fields and immutability where possible enable constant folding and aggressive optimization
- JIT shines in tight loops (runs many times, does small things each time)
- Consistent types at call sites (monomorphism) is good for inlining
- Small methods are easy to inline (method abstraction = good)
- Pre-size collections if you know them in advance!
- Use concurrent primitives instead of the synchronized keyword
- Put the common cases first in branch logic
- Use enums with numbers and integers – they are easy to optimize!

Anti-Coding for the JIT

- Boxing/unboxing is hard to optimize – obj. creation with List<Integer> vs int[]
- Try to avoid autoboxing in hot areas of your code
- Reflection is very difficult to optimize (you should be using this sparingly regardless)
- Megamorphic call sites (10+ types at a virtual call site) – more method dispatch is bad
- Very long or large methods are nigh impossible to inline – split them up so that the JIT can optimize at least some segments of code
- Don’t use exceptions in place of logic; they are expensive
- Streams are fine, but in simple cases can be expensive – use a for loop for basic tasks
- Don’t over-synchronize code that you do not have to

The JVM World

- Polyglot Ecosystem (Kotlin, Scala, Closure, etc.)
- Garbage collection (objects die young)
- Project Loom (Virtual threads, continuation)
- Project Valhalla (Value types, primitive generics)
- Class loading and the memory model
- Many other JITs!
  - PyPy (Python), V8 (JavaScript), LuaJIT, .NET (CLR)

Closing Thoughts

- The JVM is one of the coolest pieces of technology in the world, and has decades of engineering built into it
- Write clean, encapsulated, abstract code over hand-optimized code
- The JIT compiler will handle all the seemingly wasteful abstraction
- It will produce faster, more optimized code while maintaining maintainability, readability, and correctness
- Other unique VMs/Compilers: GraalVM, LLVM Ecosystem, Rust, Swift