Apiumhub

The Exceptional Performance of Kotlin

Lil' Exception Revisited

(P.S. Avoid autoboxing)

TL;DR

Circumstances should determine the solution, not preferences.

Things that appear innocuous on their own can become problems at scale.

Apiumhub

www.apiumhub.com

Passeig de Gràcia 28, 4o,
08007 Barcelona

(+34) 934 815 085

Prologue

Apiumhub

Background (pt. 1)

“Don’t use exceptions to control flow”.

Better to use if-statements and other manners to manage flow without generating an exception.

Circumstances can change!

What could/should we do if we needed to communicate a state of exception?

```
class FooService(private val fooRepo: FooRepo) {
    fun findFoo(id: Int): Foo {
        return fooRepo.getById(id)
            ?: throw FooNotFoundException("Unable to find Foo #$id")
    }
}
```

```
class FooService(private val fooRepo: FooRepo) {
    fun findFoo(id: Int): Foo? {
        return fooRepo.getById(id)
    }
}
```

Background (pt. 1)

“Don’t use exceptions to control flow”.

Better to use if-statements and other manners to manage flow without generating an exception.

Circumstances can change!

What could/should we do if we needed to communicate a state of exception?

```
class FooService(private val fooRepo: FooRepo) {
    fun findFoo(id: Int): Foo? {
        return fooRepo.getById(id)
    }
}
```

```
class FooService(private val fooRepo: FooRepo) {
    fun findFoo(id: Int): Foo {
        return fooRepo.getById(id)
            ?: throw FooNotFoundException("Unable to find Foo #$id")
    }
}
```

```
class FooService(private val fooRepo: FooRepo) {
    fun findFoo(id: Int): Result<Foo> {
        return fooRepo.getById(id)?.let { Result.success(it) }
            ?: Result.failure(FooNotFoundException("Cannot find Foo #$id"))
    }
}
```

Apiumhub

Background (pt. 2)

Based on the article “The Exceptional Performance of Lil’ Exception” by Aleksey Shipilëv.

https://shipilev.net/blog/2014/exceptional-performance/

TL;DR - Measure the performance of exception handling with a variety of techniques using JMH.

Iteratively elevate the probability of an exception occurring to show a higher and higher exception-handling load.

What if we write it in ~~Rust~~ Kotlin?

About JMH

Java Microbenchmark Harness: A structure for mounting and executing performance tests (preferably for small sections of code).

Gives options to show operation time, objects created, etc.

Avoids risks like:

- Performance distortion due to the warm-up of the JVM.
- Use of the wrong clock for measuring elapsed time.

More information:
https://www.baeldung.com/java-microbenchmark-harness

The Exceptions Family

“Lil’ Exception”: A standard exception that contains (hypothetical) metadata.

```
open class LilException : Exception {

    val metadata: Int

    constructor(metadata: Int) : super() {
        this.metadata = metadata
    }

    constructor(e: LilException, metadata: Int) : super(e) {
        this.metadata = metadata
    }

}
```

“Lil’ Stackless Exception”: Modified to not generate a stack trace on instantiation.

```
class LilStacklessException : LilException {

    constructor(metadata: Int) : super(metadata)

    constructor(e: LilStacklessException, metadata: Int) : super(e, metadata)

    // Do Nothing
    @Synchronized
    override fun fillInStackTrace(): Throwable = this

}
```

Apiumhub

More Members!

“Lil’ Result”: A sealed class that contains “success” and “failure” implementations (and could contain more).

```
sealed class LilResult {

    class LilSuccess(var result: Int): LilResult()

    class LilFailure(val metadata: Int): LilResult()

}
```

“Lil’ Outcome”: Inline class that contains the result value.

```
inline class LilOutcome(val value: Int) {
    fun isSuccess() = value != -1
}
```

Apiumhub

Getting To Work

Apiumhub

The Conditions

Launch a call in two scenarios to return either a result or an exception:

- One layer of functions (and using **`-XX:-Inline`**).
- Sixteen layers of functions.

The exception probability is incremented logarithmically (1 ppm => 900.000 ppm).

Five rounds of “warm-up”, then five rounds of execution in five sub-processes.

Execution time is measured in nanoseconds.

The Approaches: “Dynamic”

Create an exception with a stack trace and permit it to bubble up.

Advantages:

- Caught and handled only where it’s needed (i.e. try/catch).

Disadvantages:

- Generation of a stack trace for the construction of every exception.
- “Stack unwinding” - can be costly for exceptions generated in code underneath many layers of functions.

```
private fun e01(): Int = e02() * 2
private fun e02(): Int = e03() * 2
private fun e03(): Int = e04() * 2
private fun e04(): Int = e05() * 2
private fun e05(): Int = e06() * 2
private fun e06(): Int = e07() * 2
private fun e07(): Int = e08() * 2
private fun e08(): Int = e09() * 2
private fun e09(): Int = e10() * 2
private fun e10(): Int = e11() * 2
private fun e11(): Int = e12() * 2
private fun e12(): Int = e13() * 2
private fun e13(): Int = e14() * 2
private fun e14(): Int = e15() * 2
private fun e15(): Int = e16() * 2
private fun e16(): Int =
    if (ThreadLocalRandom.current().nextInt(RANDOM_RANGE) < PARTS) {
        throw LilException(source)
    } else {
        source
    }
```

Apiumhub

The Approaches: “Dynamic Stackless”

Create an exception *without* a stack trace and permit it to bubble up.

Advantages:

- Avoids the costly generation of a stack trace.

Disadvantages:

- Will not be able to use the stack trace in exception reporting.

```
private fun es01(): Int = es02() * 2
private fun es02(): Int = es03() * 2
private fun es03(): Int = es04() * 2
private fun es04(): Int = es05() * 2
private fun es05(): Int = es06() * 2
private fun es06(): Int = es07() * 2
private fun es07(): Int = es08() * 2
private fun es08(): Int = es09() * 2
private fun es09(): Int = es10() * 2
private fun es10(): Int = es11() * 2
private fun es11(): Int = es12() * 2
private fun es12(): Int = es13() * 2
private fun es13(): Int = es14() * 2
private fun es14(): Int = es15() * 2
private fun es15(): Int = es16() * 2
private fun es16(): Int =
    if (ThreadLocalRandom.current().nextInt(RANDOM_RANGE) < PARTS) {
        throw LilStacklessException(source)
    } else {
        source
    }
```

The Approaches: “Static”

Like in “Dynamic”, but using a single object in cache.

Advantages:

- Minimizes stack trace generation.
- Minimizes object creation.

Disadvantages:

- Will not be able to use the stack trace in exception reporting.
- Potential problems in thread-safety.

```
private fun s01(): Int = s02() * 2
private fun s02(): Int = s03() * 2
private fun s03(): Int = s04() * 2
private fun s04(): Int = s05() * 2
private fun s05(): Int = s06() * 2
private fun s06(): Int = s07() * 2
private fun s07(): Int = s08() * 2
private fun s08(): Int = s09() * 2
private fun s09(): Int = s10() * 2
private fun s10(): Int = s11() * 2
private fun s11(): Int = s12() * 2
private fun s12(): Int = s13() * 2
private fun s13(): Int = s14() * 2
private fun s14(): Int = s15() * 2
private fun s15(): Int = s16() * 2
private fun s16(): Int =
    if (ThreadLocalRandom.current().nextInt(RANDOM_RANGE) < PARTS) {
        throw staticException
    } else {
        source
    }
```

Apiumhub

The Approaches: “Flags”

Return a value designated as “exception case” instead of throwing an exception.

Advantages:

- Avoids the costly generation of a stack trace.
- Consistent flow.

Disadvantages:

- Verifying the exception case costs ops.
- A potential mix of concerns (i.e. a business object with the control logic).

```
private fun m01(): Int { val v = m02(); return if (v != -1) v * 2 else -1 }
private fun m02(): Int { val v = m03(); return if (v != -1) v * 2 else -1 }
private fun m03(): Int { val v = m04(); return if (v != -1) v * 2 else -1 }
private fun m04(): Int { val v = m05(); return if (v != -1) v * 2 else -1 }
private fun m05(): Int { val v = m06(); return if (v != -1) v * 2 else -1 }
private fun m06(): Int { val v = m07(); return if (v != -1) v * 2 else -1 }
private fun m07(): Int { val v = m08(); return if (v != -1) v * 2 else -1 }
private fun m08(): Int { val v = m09(); return if (v != -1) v * 2 else -1 }
private fun m09(): Int { val v = m10(); return if (v != -1) v * 2 else -1 }
private fun m10(): Int { val v = m11(); return if (v != -1) v * 2 else -1 }
private fun m11(): Int { val v = m12(); return if (v != -1) v * 2 else -1 }
private fun m12(): Int { val v = m13(); return if (v != -1) v * 2 else -1 }
private fun m13(): Int { val v = m14(); return if (v != -1) v * 2 else -1 }
private fun m14(): Int { val v = m15(); return if (v != -1) v * 2 else -1 }
private fun m15(): Int { val v = m16(); return if (v != -1) v * 2 else -1 }
private fun m16(): Int =
    if (ThreadLocalRandom.current().nextInt(RANDOM_RANGE) < PARTS) {
        -1
    } else {
        source
    }
```

The Approaches: “Sealed Class”

Designate that the function returns an object with a type of a fixed set of classes to signal either a success or a failure (i.e. a Monad).

Advantages:

- Forces that exception cases are handled without try/catch.
- Consistent flow.

Disadvantages:

- High turnover of objects due to creating one for each call to the function.

```
private fun sc01(): LilResult { val result = sc02(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc02(): LilResult { val result = sc03(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc03(): LilResult { val result = sc04(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc04(): LilResult { val result = sc05(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc05(): LilResult { val result = sc06(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc06(): LilResult { val result = sc07(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc07(): LilResult { val result = sc08(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc08(): LilResult { val result = sc09(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc09(): LilResult { val result = sc10(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc10(): LilResult { val result = sc11(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc11(): LilResult { val result = sc12(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc12(): LilResult { val result = sc13(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc13(): LilResult { val result = sc14(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc14(): LilResult { val result = sc15(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc15(): LilResult { val result = sc16(); if (result is LilResult.LilSuccess) { result.result *= 2 }; return result }
private fun sc16(): LilResult =
    if (ThreadLocalRandom.current().nextInt(RANDOM_RANGE) < PARTS) {
        LilResult.LilFailure( metadata: -1)
    } else {
        LilResult.LilSuccess(source)
    }
```

Apiumhub

The Approaches: “Clase Inline”

Wrap the value in an inline class.

The functions of the class can determine the state of exception.

Advantages:

- Can create an “intelligent flag” that contains additional functions.
- Consistent flow.

Disadvantages:

- Still mixing business and control logic.
- Only can contain one variable in comparison with sealed classes.

```
private fun ic01(): LilOutcome { val outcome = ic02(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic02(): LilOutcome { val outcome = ic03(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic03(): LilOutcome { val outcome = ic04(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic04(): LilOutcome { val outcome = ic05(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic05(): LilOutcome { val outcome = ic06(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic06(): LilOutcome { val outcome = ic07(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic07(): LilOutcome { val outcome = ic08(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic08(): LilOutcome { val outcome = ic09(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic09(): LilOutcome { val outcome = ic10(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic10(): LilOutcome { val outcome = ic11(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic11(): LilOutcome { val outcome = ic12(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic12(): LilOutcome { val outcome = ic13(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic13(): LilOutcome { val outcome = ic14(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic14(): LilOutcome { val outcome = ic15(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic15(): LilOutcome { val outcome = ic16(); return if (outcome.isSuccess()) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ic16(): LilOutcome =
    if (ThreadLocalRandom.current().nextInt(RANDOM_RANGE) < PARTS) {
        LilOutcome( value: -1)
    } else {
        LilOutcome(source)
    }
```

```
private final int ic12() {
    int outcome = this.ic13();
    return LilOutcome.isSuccess-impl(outcome) ? LilOutcome.constructor-impl(outcome) : LilOutcome.constructor-impl(-1);
}

private final int ic13() {
    int outcome = this.ic14();
    return LilOutcome.isSuccess-impl(outcome) ? LilOutcome.constructor-impl(outcome) : LilOutcome.constructor-impl(-1);
}

private final int ic14() {
    int outcome = this.ic15();
    return LilOutcome.isSuccess-impl(outcome) ? LilOutcome.constructor-impl(outcome) : LilOutcome.constructor-impl(-1);
}

private final int ic15() {
    int outcome = this.ic16();
    return LilOutcome.isSuccess-impl(outcome) ? LilOutcome.constructor-impl(outcome) : LilOutcome.constructor-impl(-1);
}

private final int ic16() {
    return ThreadLocalRandom.current().nextInt( bound: 1000000) < PARTS ? LilOutcome.constructor-impl(-1) : LilOutcome.constructor-impl(this.source);
}
```

Apiumhub

The Approaches: “Inline Autobox”

Kotlin can conduct autoboxing of inline classes in specific circumstances.

https://typealias.com/guides/inline-classes-and-autoboxing/

Advantages:

- ◯ 😀

Disadvantages:

- ◯ A lot of ops and object turnover in the heap due to the conversion of a primitive into a class and back.

```
private fun ia01(): LilOutcome? { val outcome = ia02(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia02(): LilOutcome? { val outcome = ia03(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia03(): LilOutcome? { val outcome = ia04(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia04(): LilOutcome? { val outcome = ia05(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia05(): LilOutcome? { val outcome = ia06(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia06(): LilOutcome? { val outcome = ia07(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia07(): LilOutcome? { val outcome = ia08(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia08(): LilOutcome? { val outcome = ia09(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia09(): LilOutcome? { val outcome = ia10(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia10(): LilOutcome? { val outcome = ia11(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia11(): LilOutcome? { val outcome = ia12(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia12(): LilOutcome? { val outcome = ia13(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia13(): LilOutcome? { val outcome = ia14(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia14(): LilOutcome? { val outcome = ia15(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia15(): LilOutcome? { val outcome = ia16(); return if (outcome?.isSuccess() == true) LilOutcome(outcome.value) else LilOutcome( value: -1) }
private fun ia16(): LilOutcome? =
    if (ThreadLocalRandom.current().nextInt(RANDOM_RANGE) < PARTS) {
        null
    } else {
        LilOutcome(source)
    }
```

```
private final LilOutcome ia15() {
   LilOutcome outcome = this.ia16();
   LilOutcome var10000;
   if (outcome != null) {
      if (LilOutcome.isSuccess-impl(outcome.unbox-impl())) {
         var10000 = LilOutcome.box-impl(LilOutcome.constructor-impl(outcome.unbox-impl()));
         return var10000;
      }
   }

   var10000 = LilOutcome.box-impl(LilOutcome.constructor-impl(-1));
   return var10000;
}

private final LilOutcome ia16() {
   return ThreadLocalRandom.current().nextInt( bound: 1000000) < PARTS ? null : LilOutcome.box-impl(LilOutcome.constructor-impl(this.source));
}
```

Apiumhub

The Approaches: “Result (Primitive)”

Use a class designed by Kotlin that’s similar to the Vavr class Either.

Advantages:

- Flow of inline class plus access to exception objects.
- Consistent flow.

Disadvantages:

- Lack of primitive generics signifies more autoboxing.
- Still generating an exception.

```
private fun rp01() = rp02().map { it * 2 }
private fun rp02() = rp03().map { it * 2 }
private fun rp03() = rp04().map { it * 2 }
private fun rp04() = rp05().map { it * 2 }
private fun rp05() = rp06().map { it * 2 }
private fun rp06() = rp07().map { it * 2 }
private fun rp07() = rp08().map { it * 2 }
private fun rp08() = rp09().map { it * 2 }
private fun rp09() = rp10().map { it * 2 }
private fun rp10() = rp11().map { it * 2 }
private fun rp11() = rp12().map { it * 2 }
private fun rp12() = rp13().map { it * 2 }
private fun rp13() = rp14().map { it * 2 }
private fun rp14() = rp15().map { it * 2 }
private fun rp15() = rp16().map { it * 2 }
private fun rp16(): Result<Int> {
    return if (callSucceeded()) Result.success(source) else Result.failure(LilException(source))
}
```

```
private final Object rp15_d1pmJ48/* $FF was: rp15-d1pmJ48*/() {
   Object var1 = this.rp16-d1pmJ48();
   Object var10000;
   if (Result.isSuccess-impl(var1)) {
      kotlin.Result.Companion var2 = Result.Companion;
      int it = ((Number)var1).intValue();
      int var4 = false;
      Integer var5 = it * 2;
      var10000 = Result.constructor-impl(var5);
   } else {
      var10000 = Result.constructor-impl(var1);
   }

   return var10000;
}
```

```
private final Object rp16_d1pmJ48/* $FF was: rp16-d1pmJ48*/() {
   Object var10000;
   kotlin.Result.Companion var1;
   if (this.callSucceeded()) {
      var1 = Result.Companion;
      Integer var2 = this.source;
      var10000 = Result.constructor-impl(var2);
   } else {
      var1 = Result.Companion;
      Throwable var3 = (Throwable)(new LilException(this.source));
      var10000 = Result.constructor-impl(ResultKt.createFailure(var3));
   }

   return var10000;
}
```

Apiumhub

The Approaches: “Result (w/ Wrapper)”

Create a class that serves as a wrapper for the primitive.

Operator functions simulate direct operations.

Advantages:

- “Autobox penalty” eliminated.
- Consistent flow.

Disadvantages:

- Needs repetitive code for accessing the payload.
- More code to maintain.
- Still generating an exception.

```
class ResultWrapper(private var _result: Int) {
    val result: Int
        get() = _result

    operator fun times(other: Int): ResultWrapper {
        _result *= other
        return this
    }
}
```

```
private final Object rrw15_d1pmJ48/* $FF was: rrw15-d1pmJ48*/() {
   Object var1 = this.rrw16-d1pmJ48();
   Object var10000;
   if (Result.isSuccess-impl(var1)) {
      kotlin.Result.Companion var2 = Result.Companion;
      ResultWrapper it = (ResultWrapper)var1;
      int var4 = false;
      it = it.times(2);
      var10000 = Result.constructor-impl(it);
   } else {
      var10000 = Result.constructor-impl(var1);
   }

   return var10000;
}
```

Apiumhub

Platform

Operating System: MacOS Monterrey 12.4

CPU: Intel Core i7 2,6 GHz 6-Core

Memory: 16 GB 2667 MHz DDR4

Java: Temurin JDK 18.0.1+10

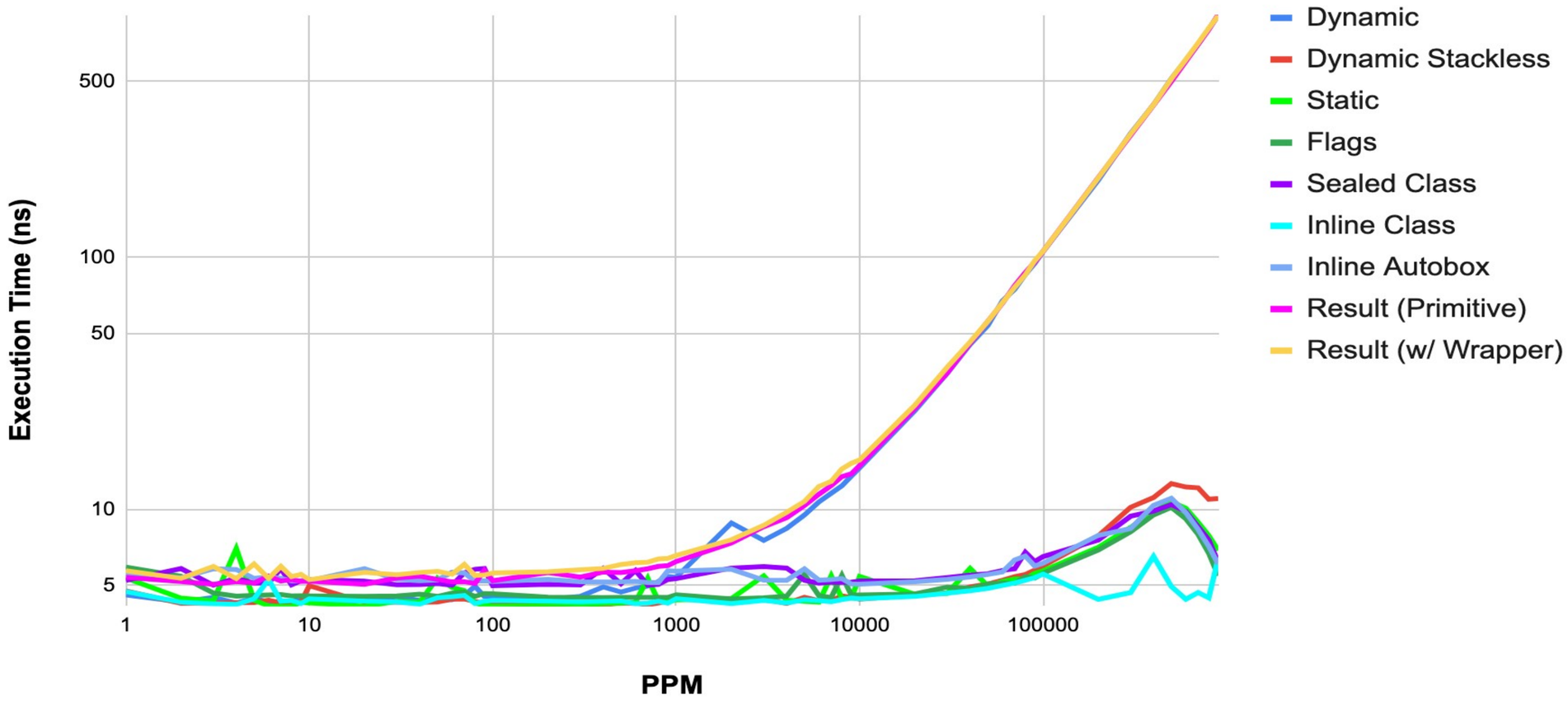
Kotlin: 1.7.0

JMH: 1.27

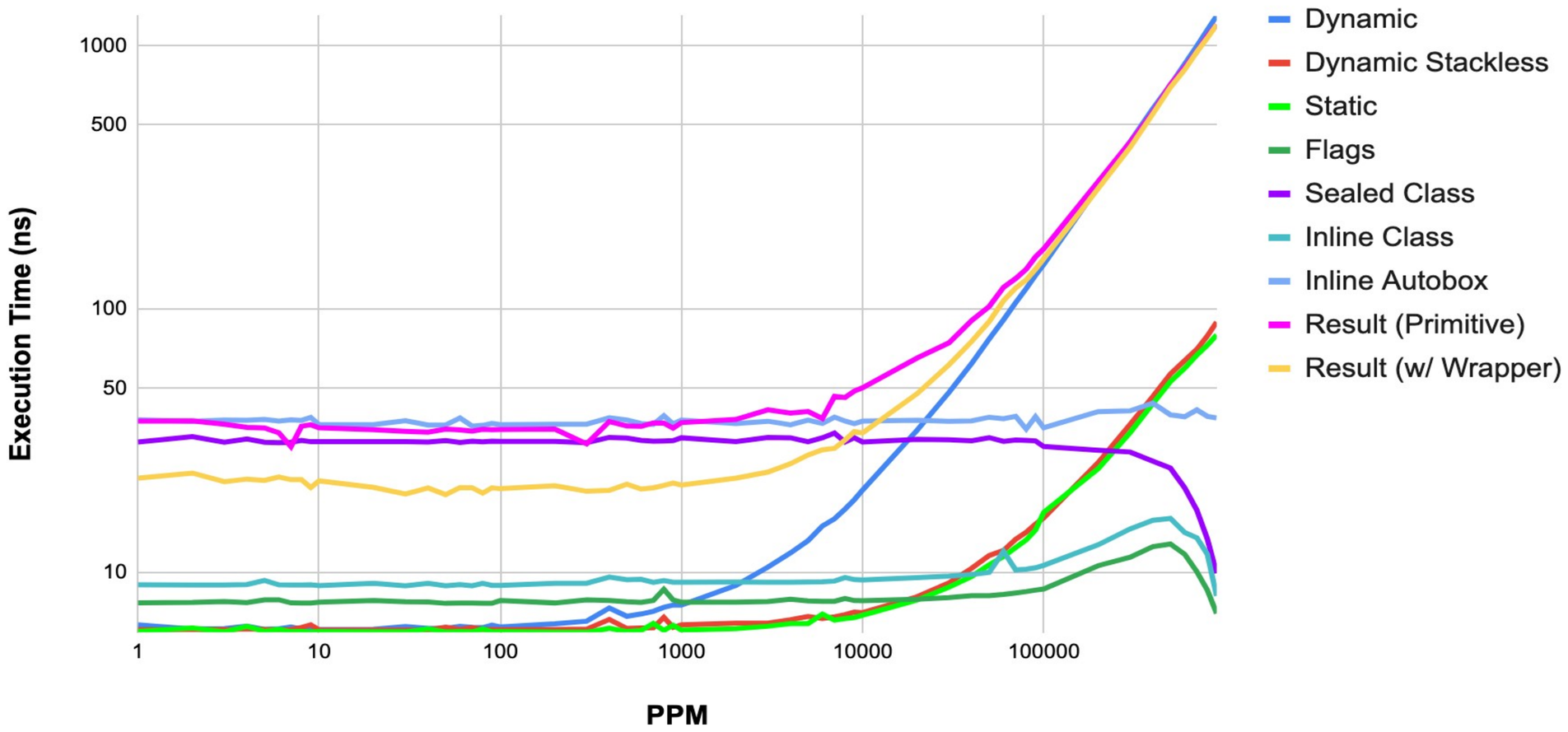
The Moment of Truth

Apiumhub

The Results - One Layer



The Results - Sixteen Layers



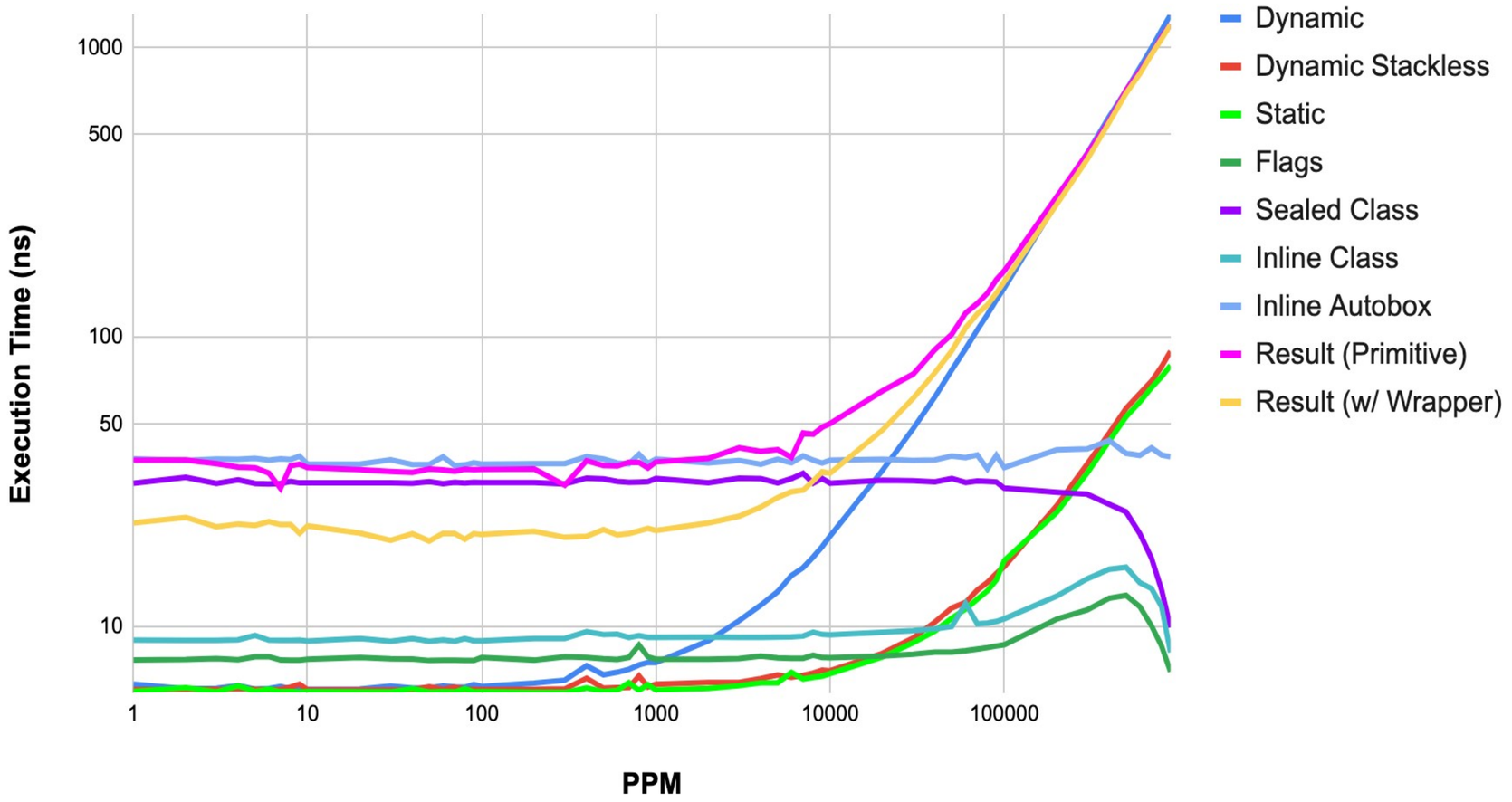
Comments

No ops < ops.

Generating a stack trace is expensive.

Result: Room for improvements.

Autoboxing is bad, mkay?



Apiumhub

Verdict?

It Depends!

Exceptions give the best performance if they’re used “exceptionally”.

Performance begins to drop off with a frequency of >0.1%.

BUT: Being I/O-constrained signifies less dependency on CPU performance.

It’d be possible to exchange performance for a more precise flow of error-handling.

Small mistakes can add up.(BE CAREFUL WITH AUTOBOXING!).

Apiumhub

www.apiumhub.com

Passeig de Gràcia 28, 4o,
08007 Barcelona

(+34) 934 815 085