KotlinConf

Idiomatic Interop

Kevin Most

Doesn't Kotlin already have 100% interop?

- Yes, but the interop can either be pleasant or clumsy
- And some features from Kotlin won't work in Java

Who should be thinking about this?

- Java library developers
- Kotlin library developers
- Anyone working in a mixed codebase

Your interop story should be

- Safe
- Performant
- Readable
- Discoverable

@JvmHappiness

`@Jvm` Annotations

- Annotations that tell the Kotlin compiler how to expose code to Java

@JvmOverloads

```kotlin
fun Date.format(
    formatString: String,
    locale: Locale = defaultLocale()
): String
```

```kotlin
format(date, "yyyyMMdd");
```

format() in **DateUtils** cannot be applied to:

Expected Parameters:		Actual Arguments:
$receiver:	**Date**	**date**
formatString:	**String**	**"yyyyMMdd"**
locale:	**Locale**	

@JvmOverloads

```kotlin
fun Date.format(
    formatString: String,
    locale: Locale = defaultLocale()
): String
```

```java
format(
        date,
        "yyyyMMdd",
        defaultLocale()
);
```

@JvmOverloads

```kotlin
@JvmOverloads
fun Date.format(
    formatString: String,
    locale: Locale = defaultLocale()
): String
```

```java
format(
        date,
        "yyyyMMdd",
        defaultLocale()
);
```

@JvmOverloads

```kotlin
@JvmOverloads
fun Date.format(
    formatString: String,
    locale: Locale = defaultLocale()
): String
```

```java
format(date, "yyyyMMdd");
```

Don't

away

```
data class User
@JvmOverloa
    val poin
)
```

```
@Nullable String id, @
@Nullable String id, @Nulla
@Nullable String id, @Nullabl
@Nullable String id, @Nullabl
@Nullable String id
<no parameters>
```

```
ble String gender, int points
llable String gender
```

"Consider a builder when faced with many constructor parameters"
- Joshua Bloch

`@JvmStatic`

- Only for use in `companion` and named `objects`
- On `fun` and `val`:
  - Generates a static method in the bytecode that delegates through to that function/property
- On `var`:
  - Also generates a static setter that delegates through

@JvmStatic

```
class User {
  companion object {
    fun create(): User
  }
}
```

```
User.Companion.create();
```

@JvmStatic

```
class User {
  companion object {
    @JvmStatic fun create(): User
  }
}
```

```
User.Companion.create();
```

@JvmStatic

```
class User {
  companion object {
    @JvmStatic fun create(): User
  }
}
```

```
User.create();
```

A less nested way?

```
class User {
  companion object {
    @JvmStatic fun create(): User
  }
}
```

```
User.create();
```

A less nested way?

```kotlin
fun create(): User

class User {
}
```

```java
User.create();
```

A less nested way?

```
fun create(): User

class User {
}
```

```
User.create();
```

A less nested way?

```
fun create(): User

class User {
}
```

```
User.create();
```

A less nested way?

```kotlin
@file:JvmName("User")

fun create(): User

class User {
}
```

```java
User.create();
```

@Throws

`@Throws`

- For Kotlin functions, property getters/setters, constructors
- Adds `throws FooException` to generated method header

@Throws

```
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    fun save(obj: T)
}
```

@Throws

```
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    fun save(obj: T)
}
```

@Throws

```
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    fun save(obj: T)
}
```

```
class UserRepository implements Repository<User> {
    @Override
    public void save(User obj) {
        throw new IOException("");
    }
}
```

@Throws

```
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    fun save(obj: T)
}
```

```
class UserRepository implements Repository<User> {
    @Override
    public void save(User obj) {
        throw new IOException("");
    }
}
```

@Throws

```
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    fun save(obj: T)
}
```

```
class UserRepository implements Repository<User> {
    @Override
    public void save(User obj) throws IOException {
        throw new IOException("");
    }
}
```

@Throws

```
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    fun save(obj: T)
}
```

```
class UserRepository implements Repository<User> {
    @Override
    public void save(User obj) throws IOException {
        throw new IOException("");
    }
}
```

@Throws

```java
class UserRepository implements Repository<User> {
    @Override
    public void save(User obj) throws IOException {
        throw new IOException("");
    }
}
```

```kotlin
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    fun save(obj: T)
}
```

@Throws

```
class UserRepository implements Repository<User> {
    @Override
    public void save(User obj) throws IOException {
        throw new IOException("");
    }
}
```

```
interface Repository<T> {
    /**
     * @throws IOException if can't save
     */
    @Throws(IOException::class)
    fun save(obj: T)
}
```

Extension functions

Extension functions

- You have a huge Java codebase with many `Utils` classes
- You add Kotlin 🎉
- You still have all these `Utils` 😧
- Awesome new Kotlin code has to call into old Java utils 😢

Extension functions

- Can we convert our Utils classes to Kotlin extensions
- While preserving their signatures in Java so existing call-sites can stay as they are?

Extension functions

```
class UserUtils {
    static boolean hasName(User user) {}
    static String getDisplayableName(User user) {}
    static boolean isAnonymous(User user) {}
    static boolean isFriendsWith(User subject, User other) {}
}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
UserUtils.isAnonymous(aUser)
UserUtils.hasName(aUser)
```

Extension functions

```
val User.hasName: Boolean get() {}
val User.displayableName: String get() {}
val User.isAnonymous: Boolean get() {}
fun User.isFriendsWith(other: User): Boolean {}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
UserUtils.isAnonymous(aUser)
UserUtils.hasName(aUser)
```

Extension functions

```
val User.hasName: Boolean get() {}
val User.displayableName: String get() {}
val User.isAnonymous: Boolean get() {}
fun User.isFriendsWith(other: User): Boolean {}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
UserUtils.isAnonymous(aUser)
UserUtils.hasName(aUser)
```

Extension functions

```
val User.hasName: Boolean get() {}
val User.displayableName: String get() {}
val User.isAnonymous: Boolean get() {}
fun User.isFriendsWith(other: User): Boolean {}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
aUser.isAnonymous
aUser.hasName
```

Extension functions

```
@file:JvmName("UserUtils") // default is UserUtilsKt


val User.hasName: Boolean get() {}
val User.displayableName: String get() {}
val User.isAnonymous: Boolean get() {}
fun User.isFriendsWith(other: User): Boolean {}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
aUser.isAnonymous
aUser.hasName
```

Extension functions

```
@file:JvmName("UserUtils") // default is UserUtilsKt


val User.hasName: Boolean get() {}
val User.displayableName: String get() {}
val User.isAnonymous: Boolean get() {}
fun User.isFriendsWith(other: User): Boolean {}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
aUser.isAnonymous
aUser.hasName
```

Extension functions

```
@file:JvmName("UserUtils") // default is UserUtilsKt


val User.hasName: Boolean @JvmName("hasName") get() {}
val User.displayableName: String get() {}
val User.isAnonymous: Boolean get() {}
fun User.isFriendsWith(other: User): Boolean {}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
aUser.isAnonymous
aUser.hasName
```

Extension functions

```
@file:JvmName("UserUtils") // default is UserUtilsKt


val User.hasName: Boolean @JvmName("hasName") get() {}
val User.displayableName: String get() {}
val User.isAnonymous: Boolean get() {}
fun User.isFriendsWith(other: User): Boolean {}
```

```
UserUtils.isAnonymous(aUser);
UserUtils.hasName(aUser);
```

```
aUser.isAnonymous
aUser.hasName
```

Inline functions

Inline functions

- Java compiler doesn't support inlining
- Can still use inline functions, but they won't be inlined
- Be mindful of performance
- **Cannot** call inline functions with reified generics from Java

Reified generics workaround

```
inline fun <reified T> View.firstViewOfType(): T? {}
```

Reified generics workaround

```
inline fun <reified T> View.firstViewOfType(): T? =
        firstViewOfType(T::class.java)

fun <T> View.firstViewOfType(type: Class<T>): T? {}
```

Visibility

Visibility

- `public`, `protected`, `private` behave as expected
- Java package-local has no equivalent in Kotlin
- Kotlin `internal` has no equivalent in Java

`internal`

- Kotlin module-level visibility
  - **Module**: IDEA, Maven, Gradle, or Ant compilation unit
- So external Java shouldn't be able to see it, right?
  - Unfortunately, `internal -> public` in bytecode

Package-local

- Java default modifier
- Only visible within the same package
- Kotlin doesn't (currently?) have a way to restrict members to the same package

`private`

- Java classes can access an inner class' `private` member
- Kotlin classes cannot

!

!

- The dreaded platform type
- Blows up when dereferenced
- Most calls into Java will return a platform type
- You should try to eliminate most/all of these in your own code
- Solution: Nullability Annotations

Nullability Annotations !

```java
interface Request<T> {
    Response<T> execute();
}
```

```java
interface Response<T> {
    T getValue();
}
```

```kotlin
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations !

```java
interface Request<T> {
    Response<T> execute();
}
```

```java
interface Response<T> {
    @Nullable T getValue();
}
```

```kotlin
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations !

- Annotating everything is super-tedious

```
kmost@kmost: ~/work/foursquare-android dev
 $ rg "@NonNull" --count --no-filename | paste -d+ -s - | bc
759

kmost@kmost: ~/work/foursquare-android dev
 $ rg "@Nullable" --count --no-filename | paste -d+ -s - | bc
475
```

Nullability Annotations !

```
static List<String> getUsers();
```

```
getUsers() // (Mutable)List<String!>!
```

Nullability Annotations !

```java
@NonNull static List<String> getUsers();
```

```
getUsers() // (Mutable)List<String!>!
```

Nullability Annotations !

```java
@NonNull static List<String> getUsers();
```

```
getUsers() // (Mutable)List<String!>
```

Nullability Annotations !

```java
@NonNull static List<@NonNull String> getUsers();
```

```
getUsers() // (Mutable)List<String!>
```

Nullability Annotations !

```java
@NonNull static List<@NonNull String> getUsers();
```

```
getUsers() // (Mutable)List<String>
```

Nullability Annotations !

```java
interface Request<T> {
    Response<T> execute();
}
```

```java
interface Response<T> {
    @Nullable T getValue();
}
```

```kotlin
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations !

```java
interface Request<T> {
    Response<@Nullable T> execute();
}
```

```java
interface Response<T> {
    T getValue();
}
```

```kotlin
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations !

```
interface Request<T> {
    Response<T> execute();
}
```

```
interface Response<@Nullable T> {
    T getValue();
}
```

```
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations !

```java
interface Request<T> {
    Response<T> execute();
}
```

```java
interface Response<T> {
    @Nullable T getValue();
}
```

```kotlin
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations !

```java
interface Request<T> {
    Response<T> execute();
}
```

```java
interface Response<T> {
    @Nullable T getValue();
    @Nullable Error<T> getError();
    @Nullable HttpResponse getRawResponse();
}
```

```
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations !

```java
interface Request<T> {
    Response<T> execute();
}
```

```java
interface Response<T> {
    @Nullable T getValue();
    @Nullable Error<@Nullable T> getError();
    @Nullable HttpResponse getRawResponse();
}
```

```kotlin
val request: Request<User> = getUser(id)
request.execute().value.displayableName
```

Nullability Annotations

Default Nullability Annotations

- Added in Kotlin 1.1.50
- Specify default annotations:
  - Per package
  - Per class
  - Per method

Default Nullability Annotations

```
dependencies {
    compile "com.google.code.findbugs:jsr305:3.0.2"
}

compileKotlin.kotlinOptions.freeCompilerArgs = [
        "-Xjsr305-annotations=enable"
]
```

Default Nullability Annotations

- JSR-305 comes with:
  - `@ParametersAreNonnullByDefault`
  - `@ParametersAreNullableByDefault`
- Annotate a package to make all parameters for all functions non-null or nullable by default

Annotating a package

package-info.java

```
@ParametersAreNonnullByDefault
package com.example.kotlinconf;
```

DIY Default Nullability Annotations

- You're not limited to `@ParametersAreNonnullByDefault`
- You can make your own nullability annotations
- Let's look at the source for `@ParametersAreNonnullByDefault`

DIY Default Nullability Annotations

ParametersAreNonnullByDefault.java

```java
@Nonnull
@TypeQualifierDefault(ElementType.PARAMETER)
public @interface ParametersAreNonnullByDefault {}
```

DIY Default Nullability Annotations

ParametersAreNonnullByDefault.java

```
@Nonnull
@TypeQualifierDefault(ElementType.PARAMETER)
public @interface ParametersAreNonnullByDefault {}
```

DIY Default Nullability Annotations

FieldsAreNonnullByDefault.java

```
@Nonnull
@TypeQualifierDefault(ElementType.FIELD)
public @interface FieldsAreNonnullByDefault {}
```

DIY Default Nullability Annotations

FieldsAreNonnullByDefault.java

```
@Nonnull
@TypeQualifierDefault(ElementType.FIELD)
public @interface FieldsAreNonnullByDefault {}
```

DIY Default Nullability Annotations

FieldsAreNullableByDefault.java

```
@Nullable
@TypeQualifierDefault(ElementType.FIELD)
public @interface FieldsAreNullableByDefault {}
```

DIY Default Nullability Annotations

FieldsAreNullableByDefault.java

```
@Nullable // not quite
@TypeQualifierDefault(ElementType.FIELD)
public @interface FieldsAreNullableByDefault {}
```

DIY Default Nullability Annotations

FieldsAreNullableByDefault.java

```
@CheckForNull
@TypeQualifierDefault(ElementType.FIELD)
public @interface FieldsAreNullableByDefault {}
```

Living the dream

package-info.java

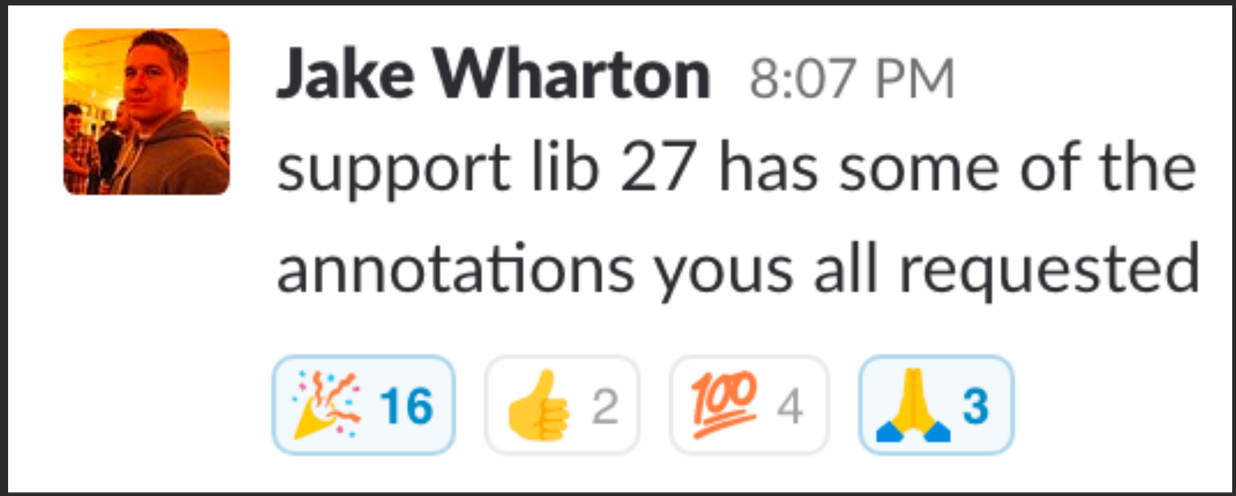
```
@ParametersAreNonnullByDefault
@FieldsAreNullableByDefault
@MethodsReturnNullableByDefault
package com.example.kotlinconf;
```

Nulls in libraries

- These solutions only work if you control the code in question
- How do you deal with Java libs that have null everywhere?
  - ex: Android

Nulls in libraries

- Ask your library maintainers



Nulls in libraries

- Submit your own PR
  - Annotations are easy and low-risk
  - Even if you "know" the nullability of members, letting the compiler enforce it for you is better

Lambdas and SAMs

Lambdas and SAMs

- Kotlin lambdas compile to a functional interface in Java
  - `() -> R` becomes `Function0<R>`
  - `(T) -> R` becomes `Function1<T, R>`
- Java SAMs compile to a special syntax in Kotlin
  - `SAMName { ... }`

SAMs

- Unfortunately, Kotlin SAMs currently don't offer SAM syntax in Kotlin
- Right now, it's best to keep your SAM types in Java

SAMs

```java
public interface JavaSAM {
    void onClick(View view);
}
```

```kotlin
val sam = JavaSAM { view -> ... }
```

SAMs

```
interface KotlinSAM {
    fun onClick(view: View)
}
```

```
val sam = JavaSAM { view -> ... }
```

SAMs

```
interface KotlinSAM {
    fun onClick(view: View)
}
```

```
val sam = KotlinSAM { view -> ... }
```

SAMs

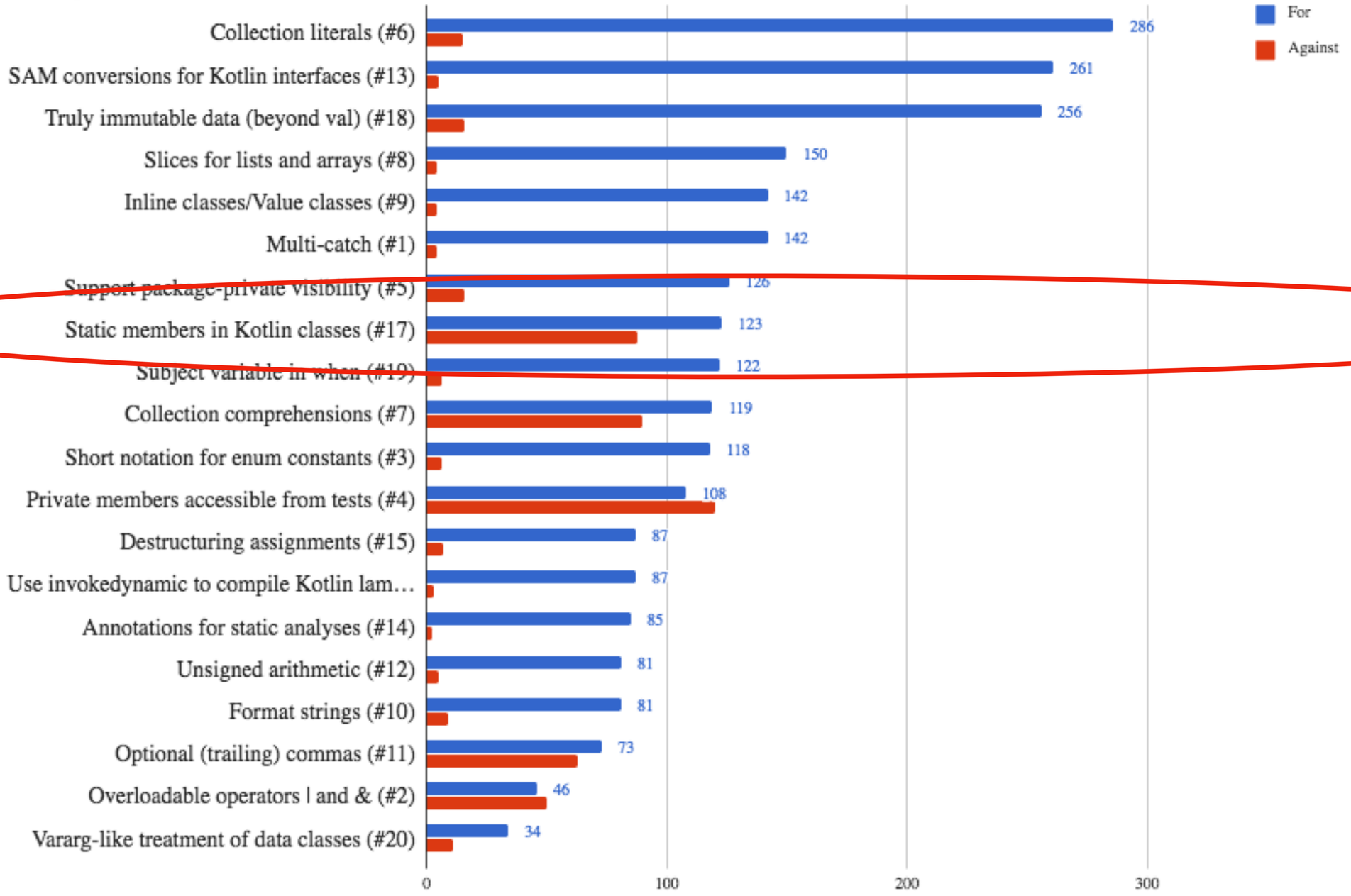
```
interface KotlinSAM {
    fun onClick(view: View)
}
```

```
val sam = KotlinSAM { view -> ... }
```

SAMs

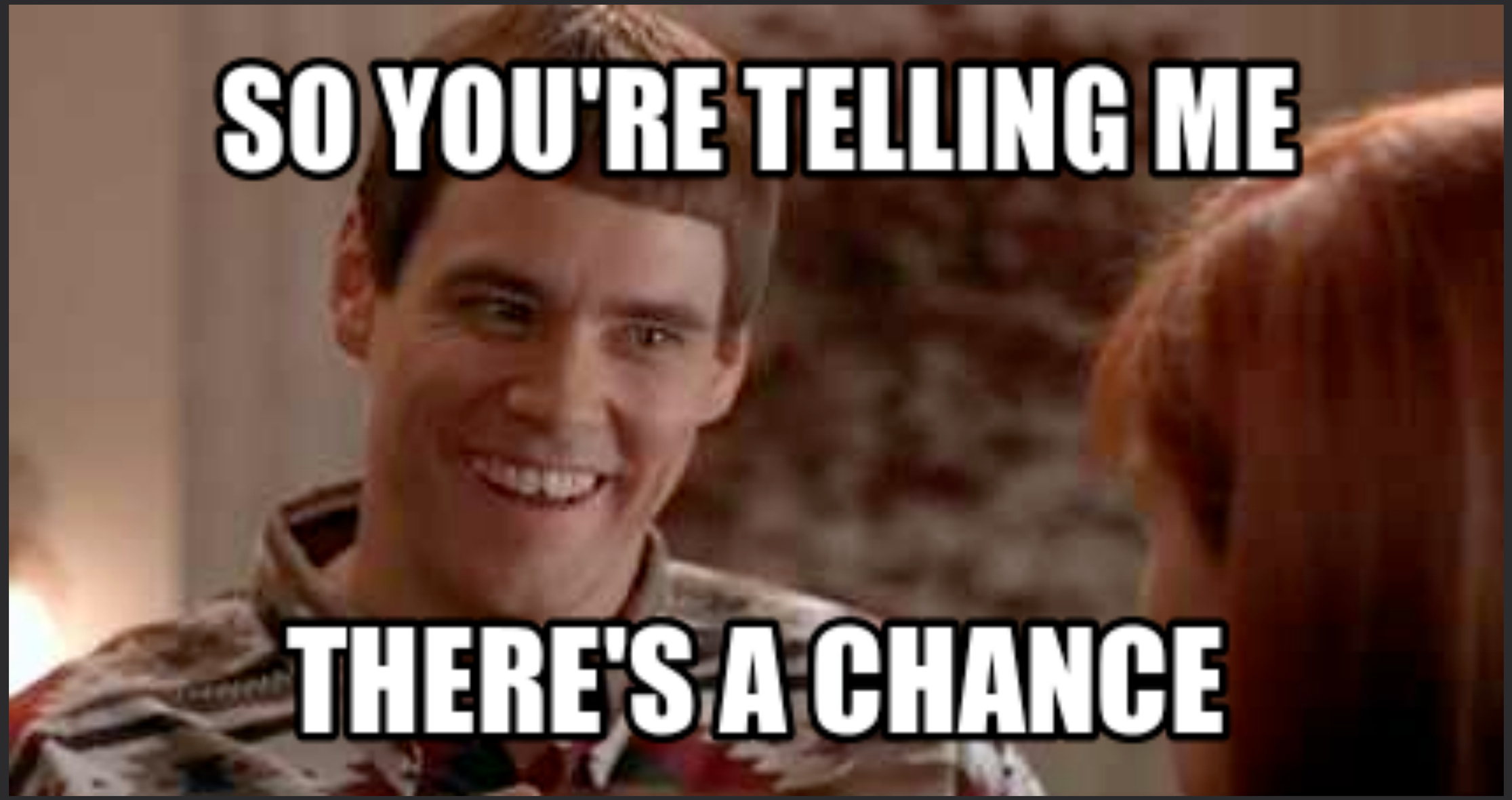
```
interface KotlinSAM {
    fun onClick(view: View)
}

val sam = object : KotlinSAM {
    override fun onClick(view: View) {
        ...
    }
}
```

Survey Results



Nominations For and Against Each Feature

"The other two [collection literals and SAM conversions] seem tractable in the foreseeable future"


SO YOU'RE TELLING ME
THERE'S A CHANCE

Lambda signatures

- Lambdas with receivers exported with receiver as 1st param
- Nullability of types is lost in Java!
- `(Foo) -> Unit` is equivalent to `Foo?.() -> Unit` from Java's perspective

Special Types

Special Types

- Most types are mapped between Java and Kotlin
- There are exceptions:
  - `Unit`
  - `Nothing`

`Unit`

- `Unit` can be mapped to `void` in most cases in Java
- It cannot, however, if `Unit` is the selected type for a generic
  - Ex: Lambdas. Java signature `FunctionN<Args, Unit>`
  - Java has to: `return Unit.INSTANCE;`

`Nothing`

- `Nothing` is the subtype of all other types
- No instances exist, not even `null`
  - So a `Nothing` function can never return; must throw/loop
- No type exists like this in Java

`Nothing`

- Generics of `Nothing` become raw types
  - `List<Nothing>` in Kotlin becomes `List` in Java
- Actual `Nothings` become `Void`
  - Consumers just have to know the method will never return

Wildcards

Wildcards

- In Java, all use-sites of a generic need to say if they are:
  - Covariant: `? extends Foo`
  - Contravariant: `? super Foo`
- In Kotlin, you just put that declaration on the type param itself:
  - Covariant: `out T`
  - Contravariant: `in T`

Wildcards

```
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<Number>> boxes = new ArrayList<>();
boxes.add(boxedInt);
```

Wildcards

```
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<Number>> boxes = new ArrayList<>();
boxes.add(boxedInt);
```

Wildcards

```
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<? extends Number>> boxes = new ArrayList<>();
boxes.add(boxedInt);
```

Wildcards

```
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<? extends Number>> boxes = new ArrayList<>();
boxes.add(boxedInt);
```

Wildcards

```
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<? extends Number>> boxes;
boxes.add(boxedInt);
```

Wildcards

```java
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<? extends Number>> boxes;
boxes.add(boxedInt);
```

```kotlin
class Box<T>(val foo: T)

fun <T> box(unboxed: T)
    = Box<T>(unboxed)
fun <T> unbox(box: Box<T>)
    = box.foo

val boxedInt: Box<Int> = box(3)
val boxed =
    mutableListOf<Box<Number>>()
boxed.add(boxedInt)
```

Wildcards

```java
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<? extends Number>> boxes;
boxes.add(boxedInt);
```

```kotlin
class Box<T>(val foo: T)

fun <T> box(unboxed: T)
    = Box<T>(unboxed)
fun <T> unbox(box: Box<T>)
    = box.foo

val boxedInt: Box<Int> = box(3)
val boxed =
    mutableListOf<Box<Number>>()
boxed.add(boxedInt)
```

Wildcards

```java
class Box<T> {
    T foo;
}

<T> Box<T> box(T unboxed) {
    return new Box<>(unboxed);
}

<T> T unbox(Box<T> box) {
    return box.foo;
}

Box<Integer> boxedInt = box(3);
List<Box<? extends Number>> boxes;
boxes.add(boxedInt);
```

```kotlin
class Box<out T>(val foo: T)

fun <T> box(unboxed: T)
    = Box<T>(unboxed)
fun <T> unbox(box: Box<T>)
    = box.foo

val boxedInt: Box<Int> = box(3)
val boxed =
    mutableListOf<Box<Number>>()
boxed.add(boxedInt)
```

Wildcards

- So `out` is roughly equivalent to `extends`
- And `in` is roughly equivalent to `super`
- Kotlin "fakes" declaration-site variance for Java by generating wildcards for all variant generics in **parameters**
- Return types remain invariant
- Final covariant types remain invariant

Wildcards

- To override the default generic behavior:
  - `@JvmWildcard` if you want variance where there is none
  - `@JvmSuppressWildcards` if you don't want variance

Data classes

Data classes

- Tuple-like classes; properties declared in constructor
- Auto-generation of `hashCode()`, `equals()`, `toString()`
  - These work perfectly in Java
- Auto-generation of `copy(...)`
  - This works *okay* in Java
  - Lack of default + named params makes it clunky

Data classes

```kotlin
data class User(
        val id: String? = null,
        val name: String? = null,
        val username: String? = null,
        val gender: String? = null,
        val points: Int = 0
)
```

```java
u.copy(u.getId(), u.getName(), u.getUsername(),
    u.getGender(), u.getPoints() + 1);
```

Data classes

```
data class User(
        val id: String? = null,
        val name: String? = null,
        val username: String? = null,
        val gender: String? = null,
        val points: Int = 0
)
```

```kotlin
data class User(
    val id: String? = null,
    val name: String? = null,
    val username: String? = null,
    val gender: String? = null,
    val points: Int = 0
) {
  fun toBuilder() = Builder(this)

  class Builder(private var user: User = User()) {
    fun id(id: String?) = apply { user = user.copy(id = id) }
    fun name(name: String?) = apply { user = user.copy(name = name
    // ...
    fun build() = user
  }
}
```

"Consider a builder when faced with many constructor parameters"
- Joshua Bloch

```
data class User(
    val id: String? = null,
    val name: String? = null,
    val username: String? = null,
    val gender: String? = null,
    val points: Int = 0
) {
  fun toBuilder() = Builder(this)

  class Builder(private var user: User = User()) {
    fun id(id: String?) = apply { user = user.copy(id = id) }
    fun name(name: String?) = apply { user = user.copy(name = name
    // ...
    fun build() = user
  }
}
```

```
data class User(
    val id: String? = null,
    val name: String? = null,
    val username: String? = null,
    val gender: String? = null,
    val points: Int = 0
) {
  fun toBuilder() = Builder(this)

  class Builder(private var user: User = User()) {
    fun id(id: String?) = apply { user = user.copy(id = id) }
    fun name(name: String?) = apply { user = user.copy(name = name
    // ...
    fun build() = user
  }
}
```

```
data class User(
    val id: String? = null,
    val name: String? = null,
    val username: String? = null,
    val gender: String? = null,
    val points: Int = 0
) {
  fun toBuilder() = Builder(this)

  class Builder(private var user: User = User()) {
    fun id(id: String?) = apply { user = user.copy(id = id) }
    fun name(name: String?) = apply { user = user.copy(name = name
    // ...
    fun build() = user
  }
}
```

```
data class User(
    val id: String? = null,
    val name: String? = null,
    val username: String? = null,
    val gender: String? = null,
    val points: Int = 0
) {
  fun toBuilder() = Builder(this)

  class Builder(private var user: User = User()) {
    fun id(id: String?) = apply { user = user.copy(id = id) }
    fun name(name: String?) = apply { user = user.copy(name = name
    // ...
    fun build() = user
  }
}
```

Overall Interop Thoughts

Overall Interop Thoughts

- `kt` ❤️ `java`
- Most of the time, interop Just Works™
- But when writing non-`private` members, say to yourself:
  - When writing Kotlin: "How will this look in Java?"
  - When writing Java: "How will this look in Kotlin?"

Quick Tip

Quick Tip

- Write (at least some) tests in the other language
  - If you use Java, write some Kotlin tests
  - If you write Kotlin, write some Java tests
- Gives you insight into the ergonomics of your public API

Resources

- Calling Kotlin from Java: https://kotlinlang.org/docs/reference/java-to-kotlin-interop.html
- Calling Java from Kotlin: https://kotlinlang.org/docs/reference/java-interop.html

Thank you!

Kevin Most

#kotlinconf17