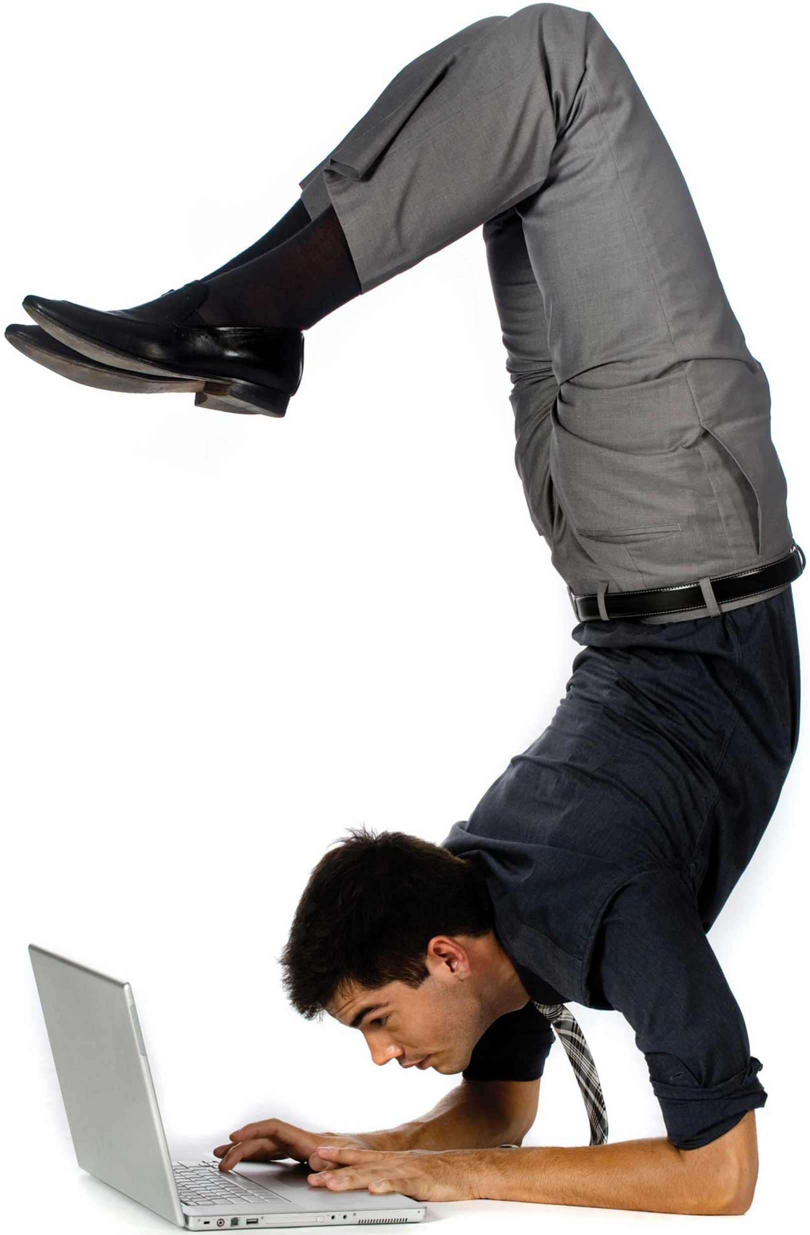
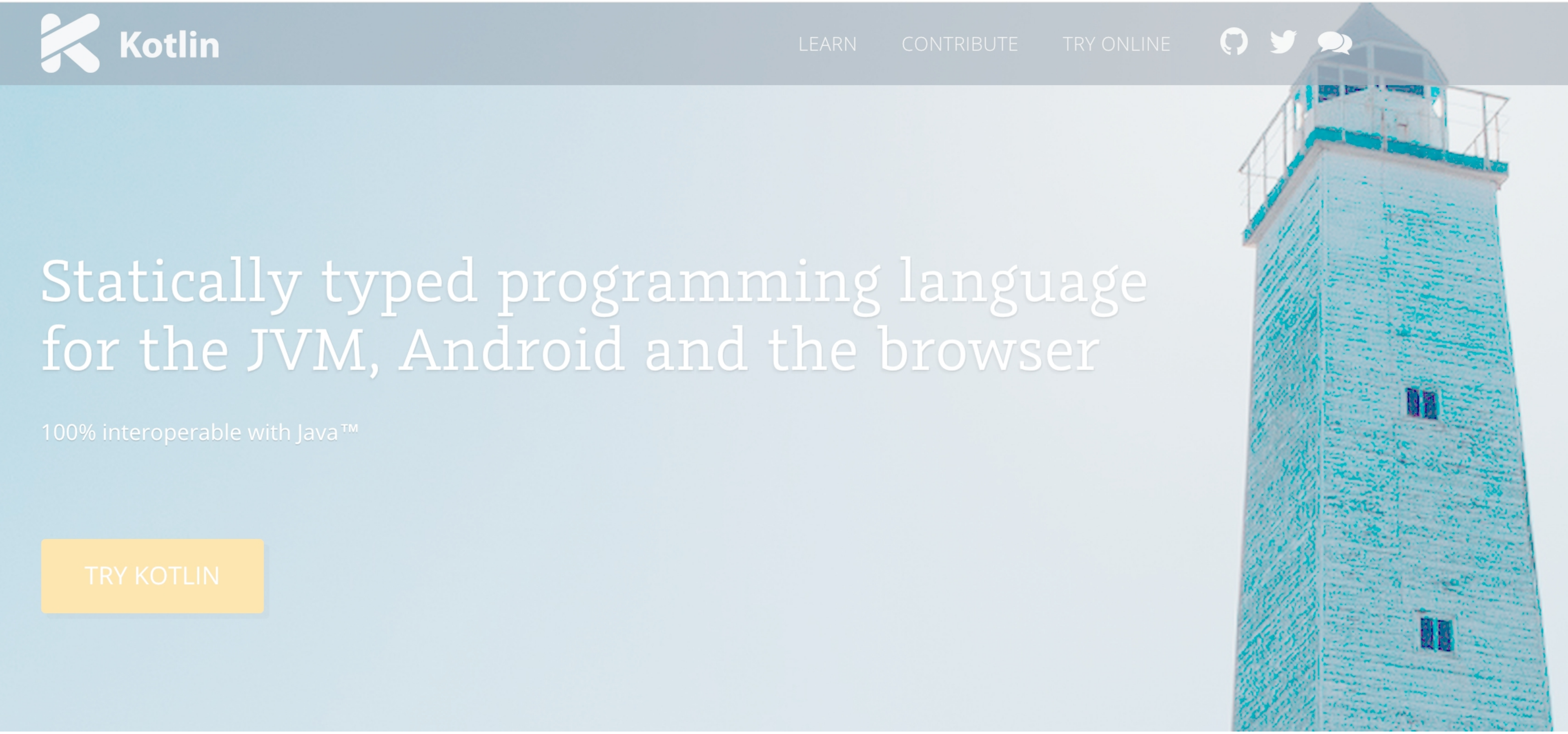


Flexible Types in Kotlin

Andrey Breslav

JetBrains

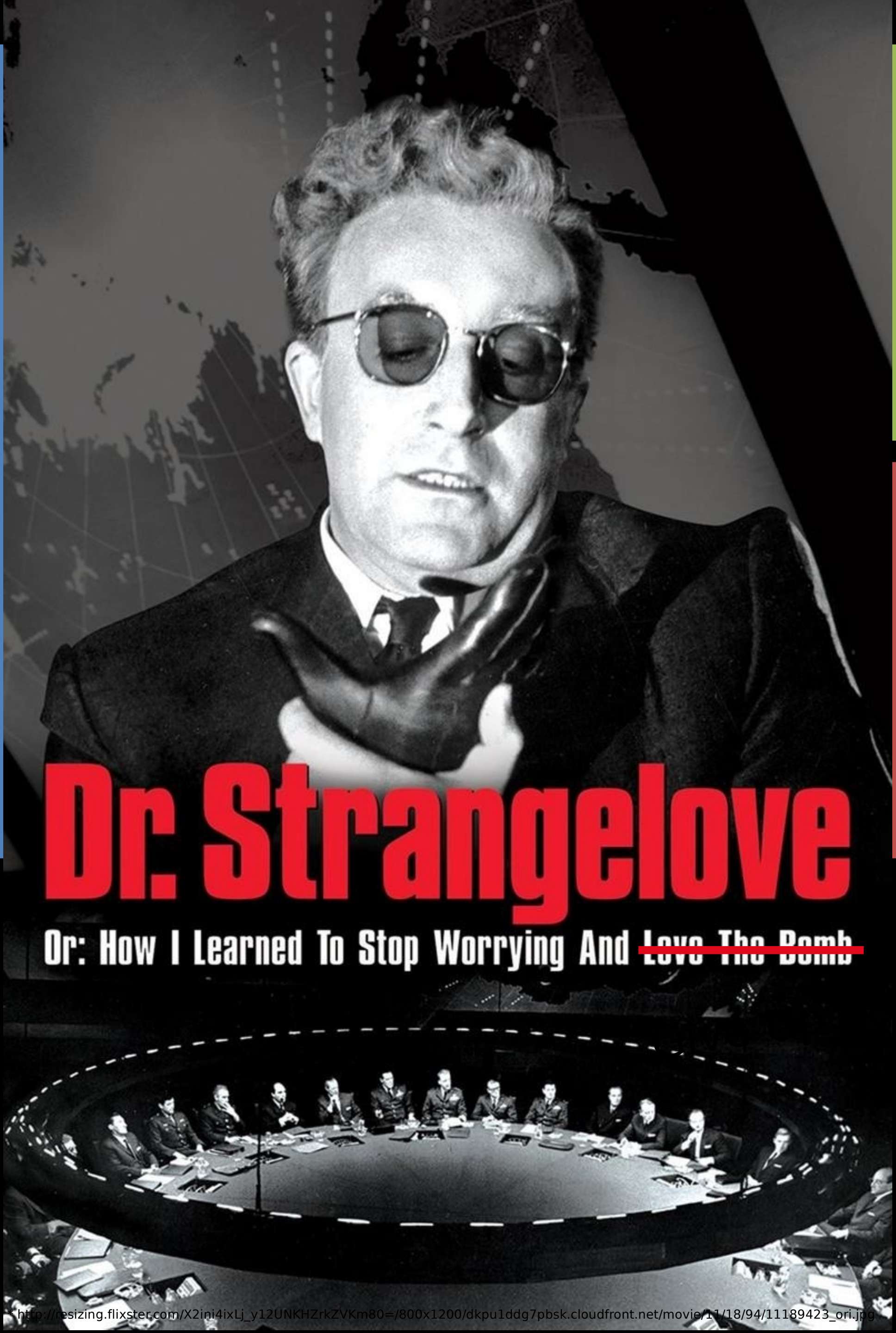
* http://www.v3.co.uk/IMG/686/292686/flexible-working-1.jpg



Statically typed programming language for the JVM, Android and the browser

100% interoperable with Java™

http://kotlinlang.org



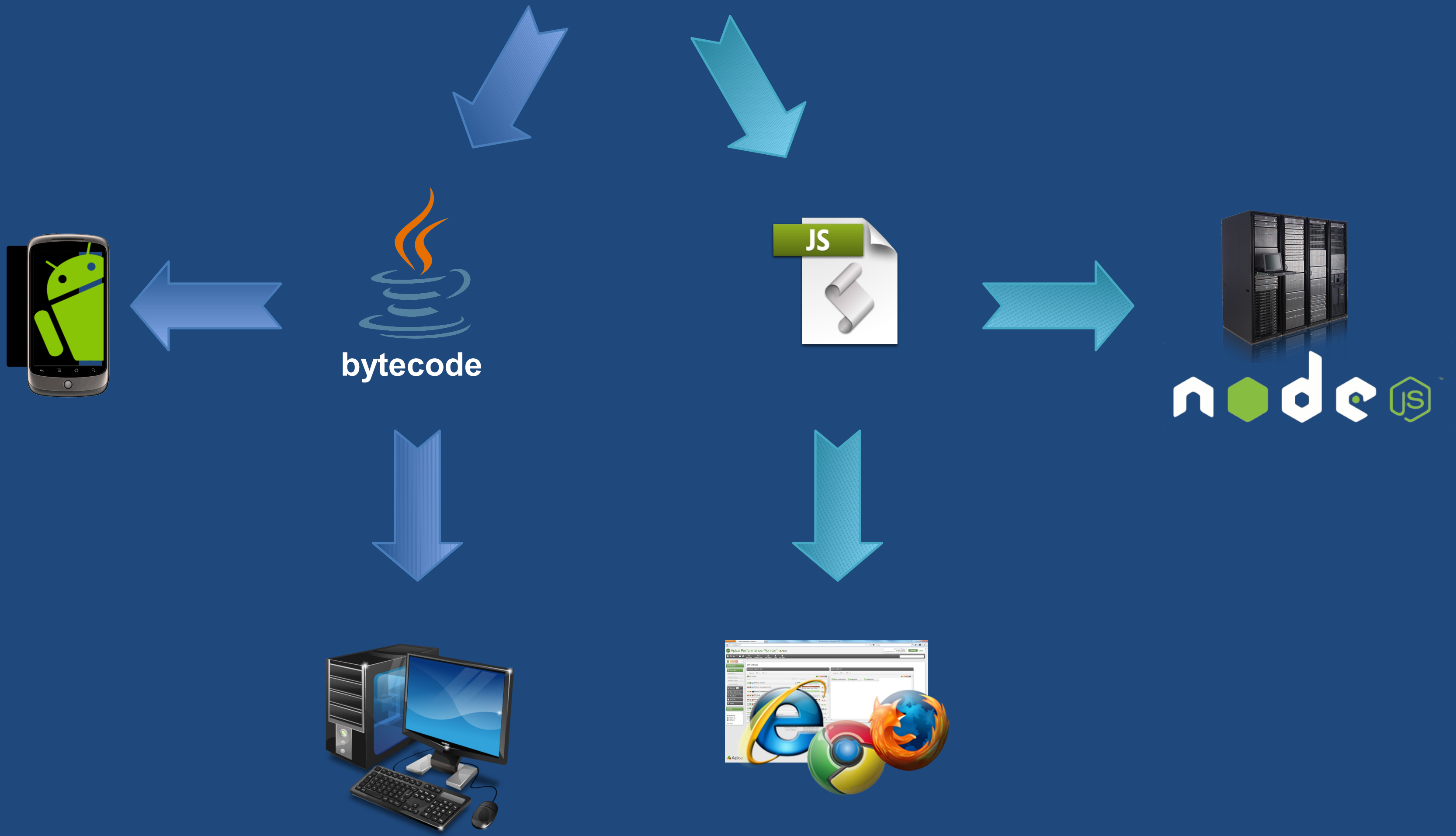
http://resizing.flixster.com/X2ini4ixLj_y12UNKHZrkZVKm80=/800x1200/dkpu1ddg7pbsk.cloudfront.net/movie/11/18/94/11189423_ori.jpg

Is the glass **full**?

Practicall y, it’s full
THE ENGINEER

http://examinetheglass.com/wp-content/uploads/2011/08/glass-of-water1.jpg
http://glsindia.com/wp-content/uploads/2013/08/team_fortress_2__the_engineer_by_onosaka_yuha-d47vyqv1.jpg

Kotlin



bytecode

Safety Features

* http://www.howtovanish.com/wp-content/uploads/2011/01/Safe.jpg

- Nullable Types
- Read-Only Collections
- Invariant Arrays
- Raw Types

* http://www.ashburnham-insurance.co.uk/wordpress/media/water-leak-from-upstairs-flat.jpg

Abstractions Leak

- A Java-compatible* language can not be safer than Java

* Depends on what you mean by “compatible”.
We are looking for a really smooth,
no-overhead interop

wpengine.netdna-cdn.com/wp-content/uploads/2013/01/integration.jpg

Interoperability

- Kotlin can call Java
  - No overhead in code
  - No runtime overhead
- No preventing intended uses of Java APIs

Example: Arrays

```
// Java
class Util {
  public static int countNonNull(Object[] arr)
{…}
}


// Kotlin
val strings = arrayOf(“a”, “b”)
Util.countNonNull(strings)
```

Interoperability

wpengine.netdna-cdn.com/wp-content/uploads/2013/01/integration.jpg

- Kotlin can call Java
  - No overhead in code
  - No runtime overhead
- No preventing intended uses of Java APIs

- Pure Kotlin’s safety should not be compromised

Nulls

- The Ultimate Disa

* http://www.maciejratajski.com/sites/default/files/work/image/ratajski-knowledge-1.jpg

```
String s = null;

s.length();
```

At Runtime

```
val s: String = null
s.length()
```

Nullabl
e
type

```
val s: String? = null
s.length()
```

At Compile Time

Check and use

```
val s: String? = …
if (s != null) {
    s.length()
}
```

Check and exit

```
if (s == null) return

s.length()
```

Rock’n’Roll

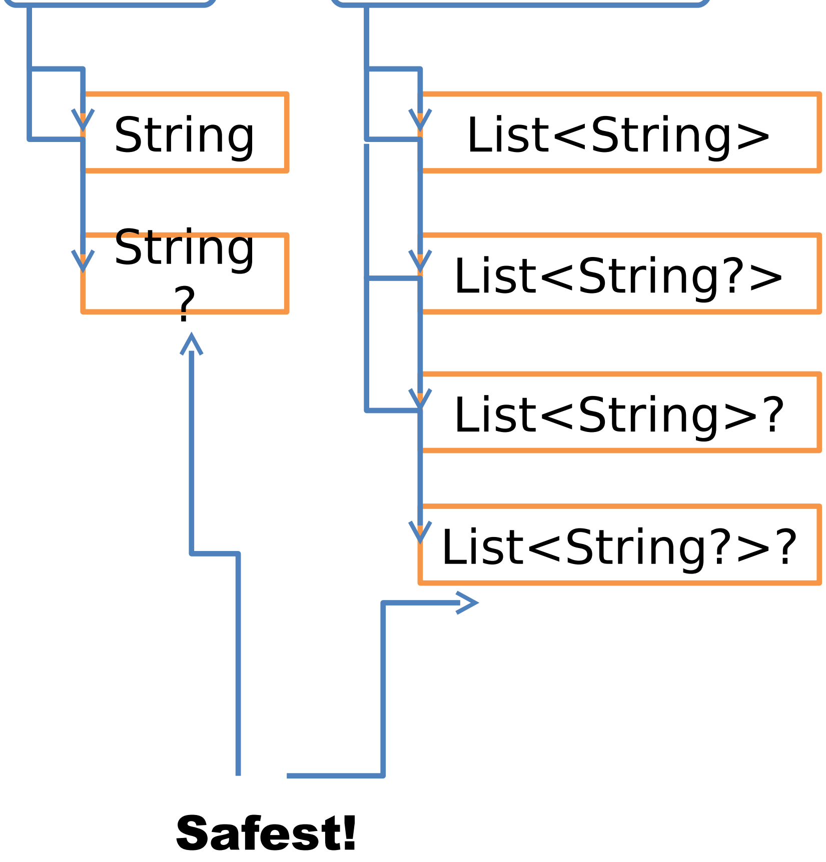
```
s?.length()
s!!.length()
(s ?: “…”).length()
```

Kotlin is **good** with **null**s

What about Java?

Java (as seen from Kotlin)

```
public class JavaClass {
    public String foo(List<String> l) {…}
}
```



String

String
?

List<String>

List<String?>

List<String>?

List<String?>?

Safest!

Java Interop: All Nullable

```
javaValue.toString().length() + 1
```

```
javaValue?.toString()?.length()!! + 1
```

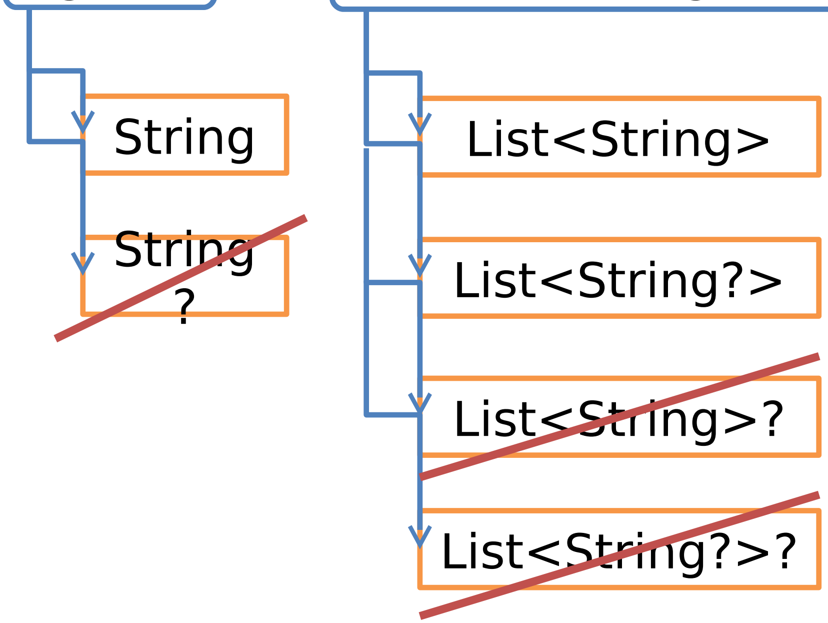
```
val l: List<String> = javaValue?.getList()!!
```

ArrayList<String?>

Your safest option doesn't work!

Annotations

```
public class JavaClass {
   @NotNull
   public String foo(@NotNull List<String> l) {…}
}
```



String

~~String?~~

List<String>

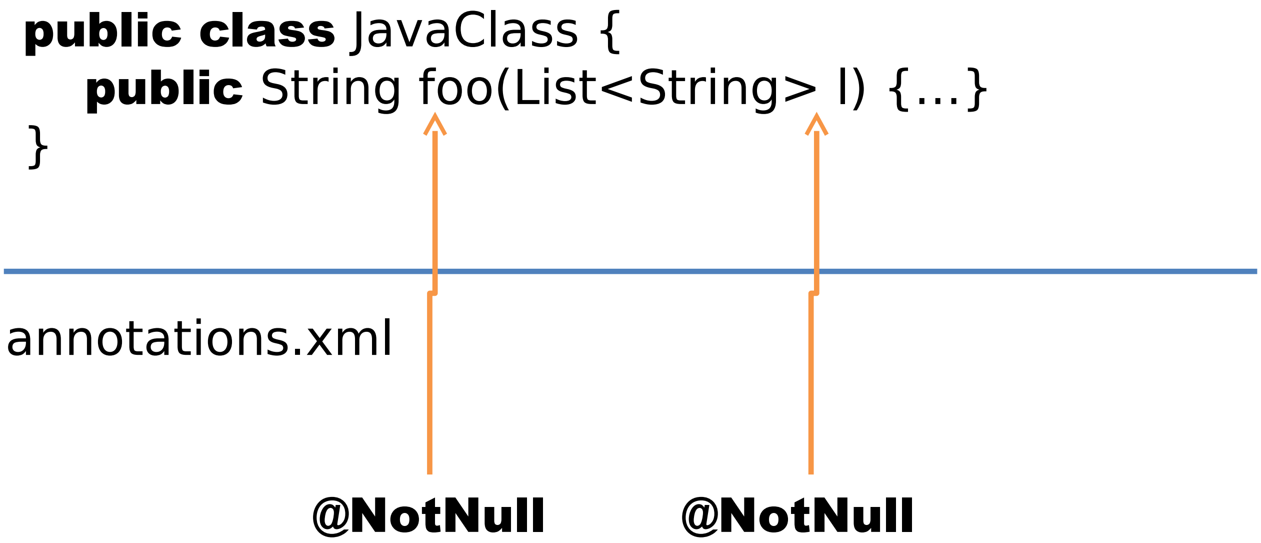
List<String?>

~~List<String>?~~

~~List<String?>?~~

Annotations are cumbersome AND **don't really help!**

External Annotations

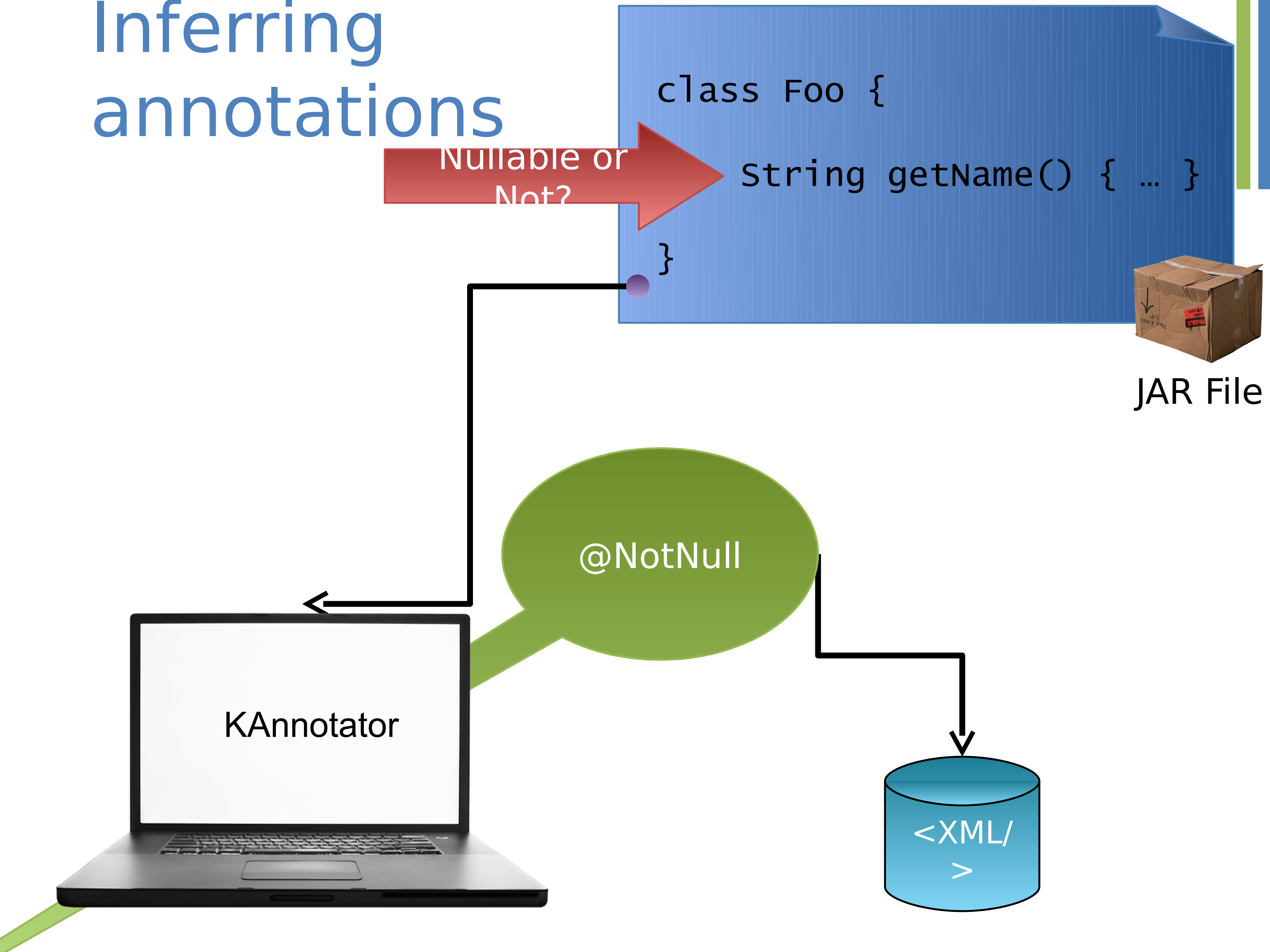


```
public class JavaClass {
    public String foo(List<String> l) {…}
}
```

annotations.xml

@NotNull **@NotNull**

Inferring annotations



```
class Foo {
    String getName() { … }
}
```

Inferring Annotations

- Well-known problem
  - A little out of fashion
  - Best tool by 2013: Julia by F. Spoto of Università di Verona
  - Now: IntelliJ IDEA has **on-the-fly inference**
- Challenge:
  - Good open source implementation
  - Support for Generic Types
  - Performance

Pick Two

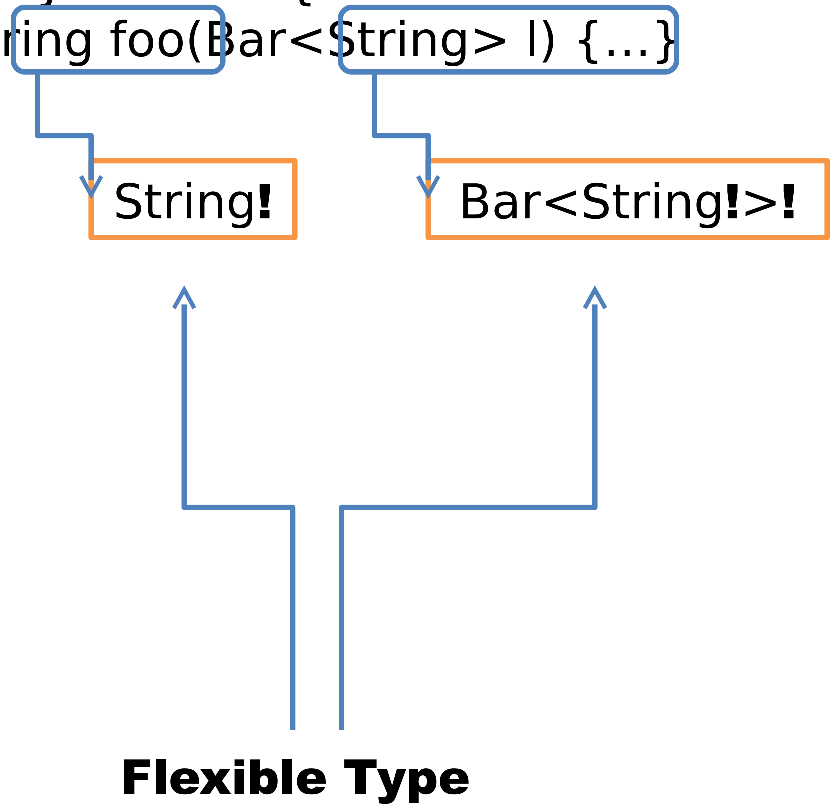
- ~~Null-safety~~
- Convenience
- Java Interop

Flexible Types!

Thanks to
Dr. Ross Tate of

Java: Flexible Types

```
public class JavaClass {
    public String foo(Bar<String> l) {…}
}
```



String!

Bar<String!>!

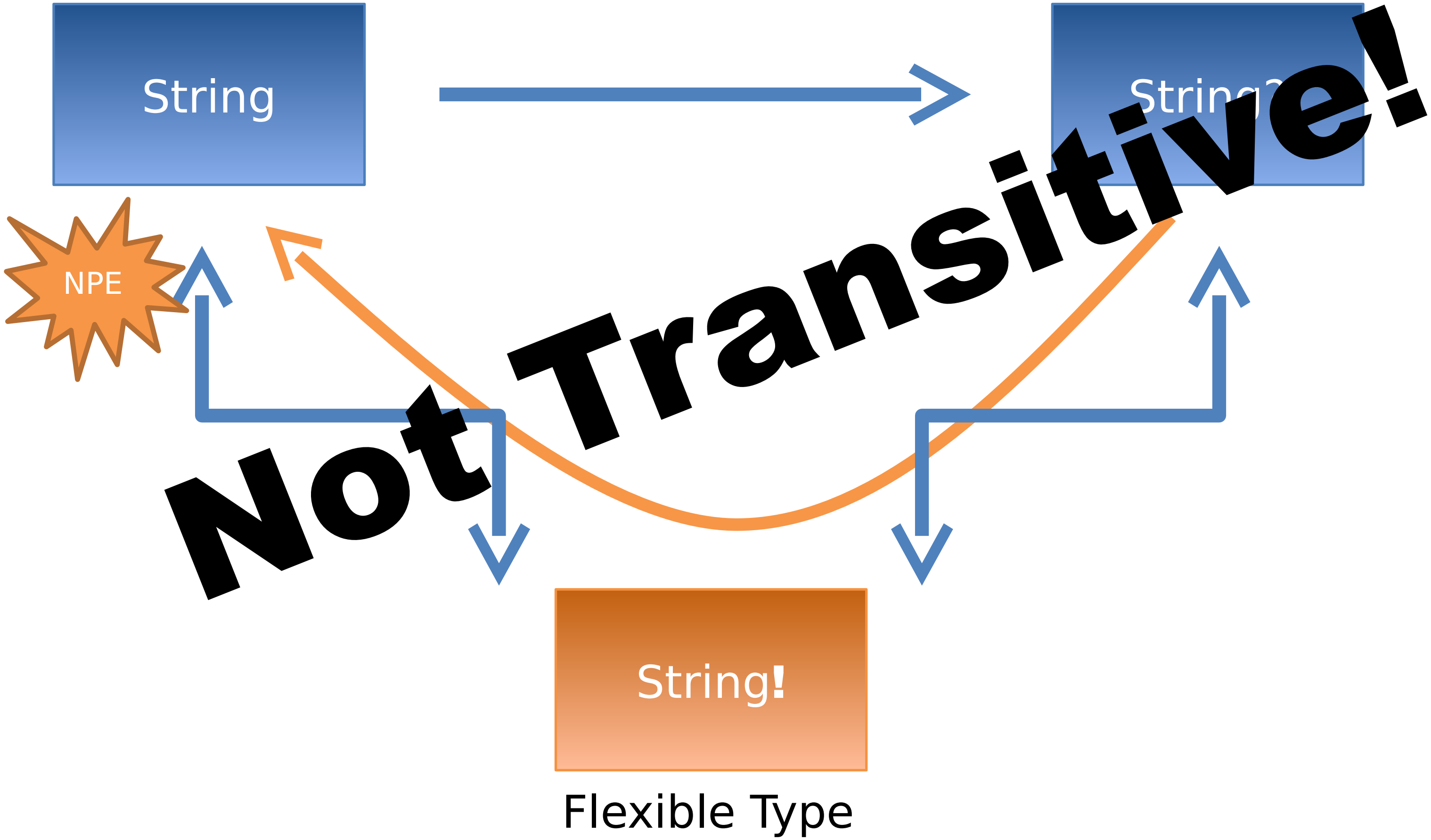
Flexible Type

+

Dereferencing Flexible Values

s: String	s: String?	s: String!
s.length()	s.length()	s.length() NPE
s?.length()	s?.length()	s?.length()
s!!.length()	s!!.length()	s!!.length()

+ Assignability



String

String?

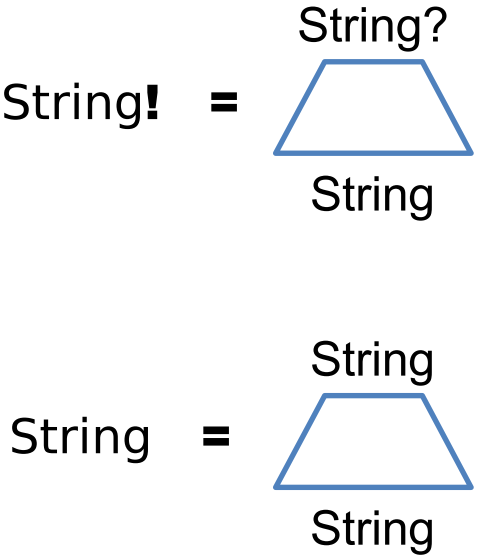
NPE

String!

Flexible Type

Not Transitive!

Representation



String?

String**!** =

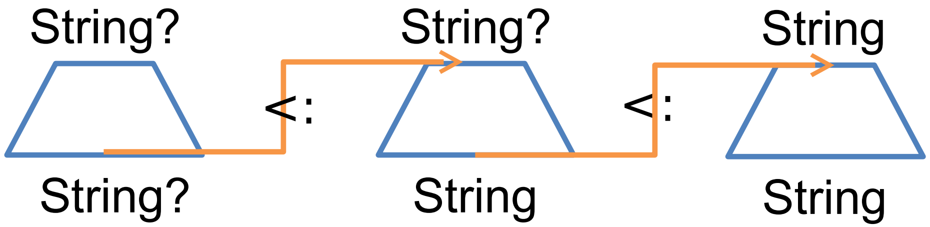
String

String

String =

String

Optimistic Subtyping:
picks the **most convenient** end

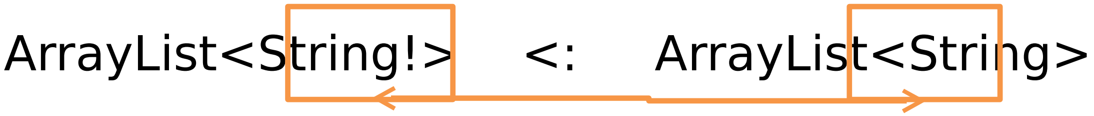


String? String? String

<: <:

String? String String

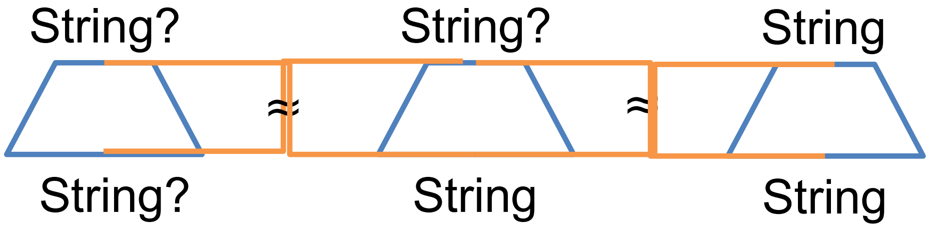
“Equivalence”



ArrayList<String!> <: ArrayList<String>

“equivalent”

$$A \approx B \quad \Leftrightarrow \quad A <: B \quad \& \quad B <: A$$



- Reflexive
- Symmetric
- **NOT** Transitive

Runtime Checks

(and their absence)

- `String <- String!`
  - Fails if the value is **null**
- `List<String> <- ArrayList<String!>!`
  - Fails if the list is **null**
  - Passes if an element is **null**

* http://img06.deviantart.net/1744/i/2013/082/5/b/there_s_a_hole_in_the_fence_where_i_used_to_see__by_tricyclemishap-d5z1u3w.jp

Some Notes

- Flexible Types are **Not Denotable!**
  - **String!** is notation, not syntax
- **Pure Kotlin is Null-Safe**
- Kotlin+Java is **as safe as Java**

- Annotations Still Applicable
  - **@NotNull String** in Java becomes **String** in Kotlin

There are many Java APIs that talk about arrays, which will never change

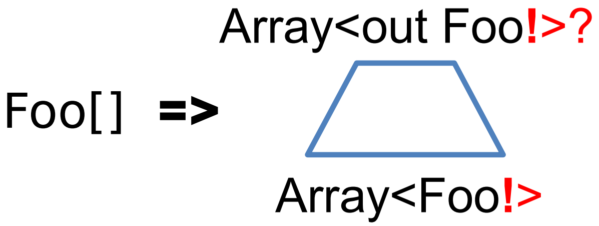
-- John Rose

Arrays

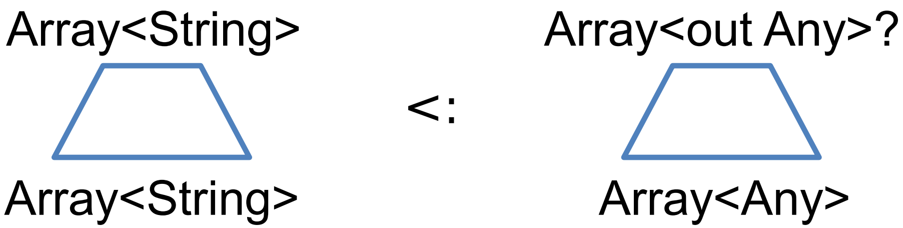
* http://www.performing-musician.com/pm/feb09/images/TechNotes_03_WideLine_array.jpg

Kotlin Arrays

- **Array<T>** is invariant
  - **Array<String>** >:< **Array<Any>**
- **Array<out Any>** is a *covariant projection*
  - similar to **Array<? extends Any>**
  - **Array<T>** <: **Array<out Any>**
  - Can’t call **arr.set(x)**



Arrays: Example



Array<String>

Array<String>

<:

Array<out Any>?

Array<Any>

Collections

```
interface List<E> {

   E get(int index);

   E set(int index, E value);

}



/** read-only */
List<Foo> getFoos() {
    return unmodifiableList(l);
}
```

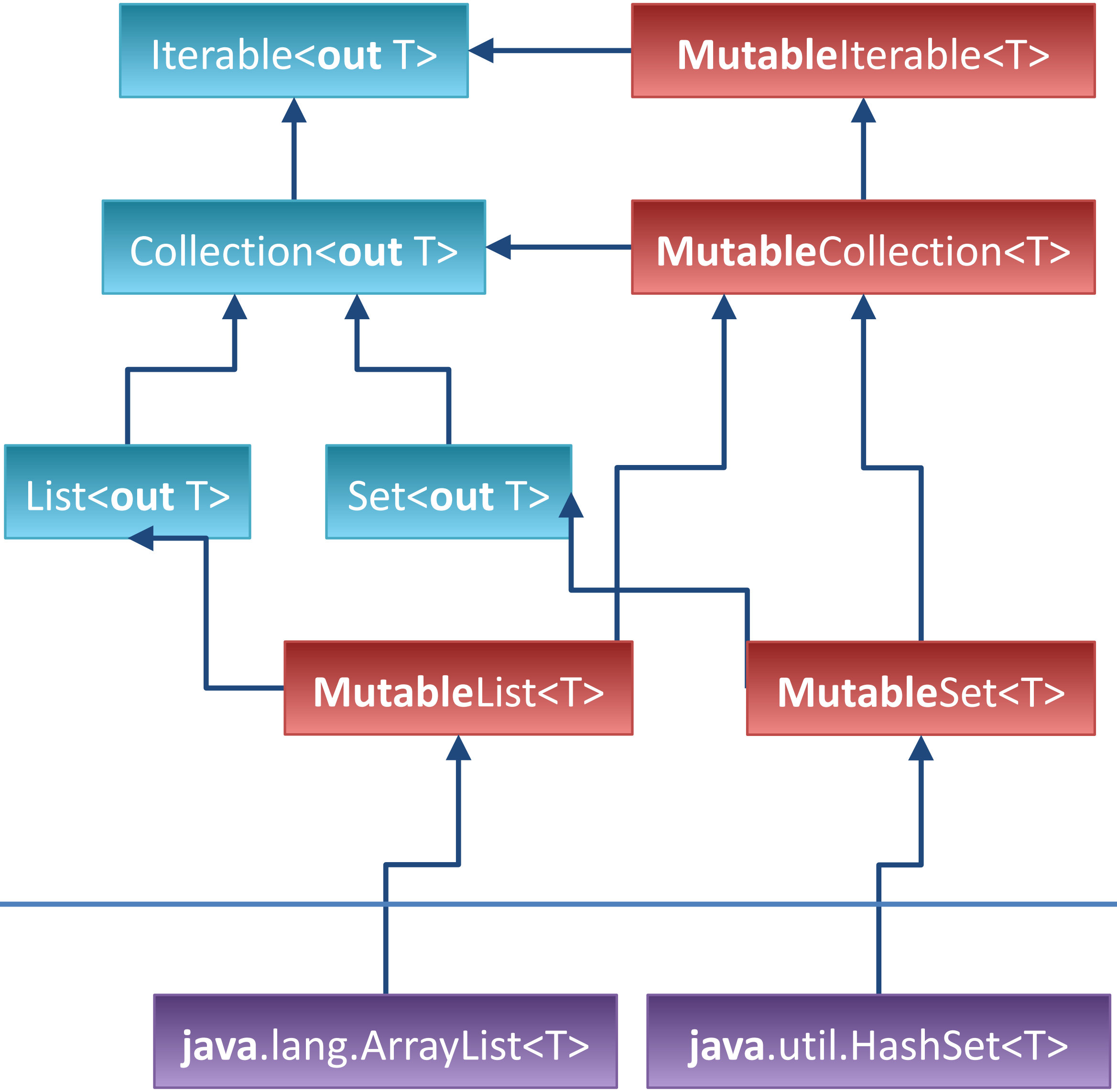
At Runtime

```
interface List<E> {
   E get(int index);
}

interface MutableList<E>
             extends List<E> {
   E set(int index, E value);
}


List<Foo> getFoos() {
    return l; // may be mutable
}
```

At Compile Time

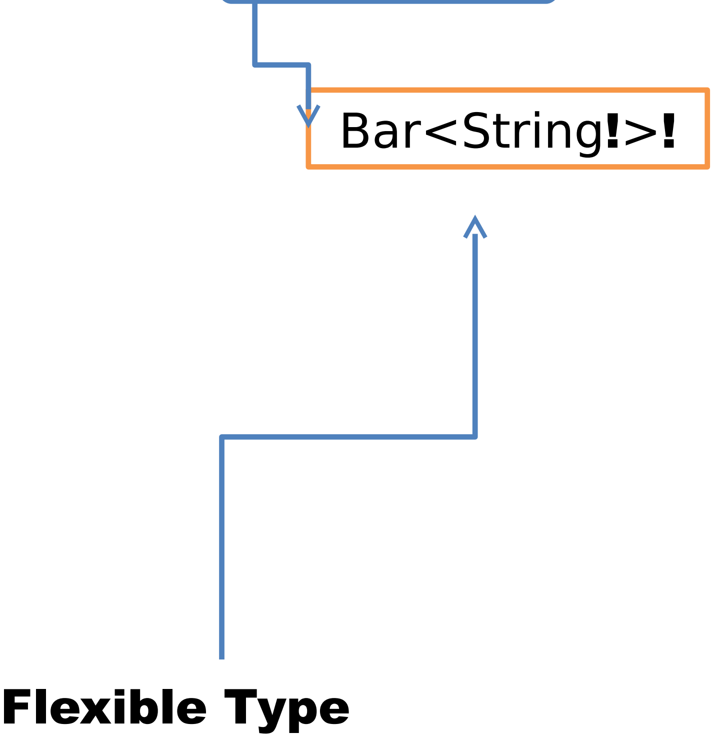
Kotlin Collections

JDK Collections



Java: Flexible Types

```
public class JavaClass {
    public String foo(Bar<String> l) {…}
}
```

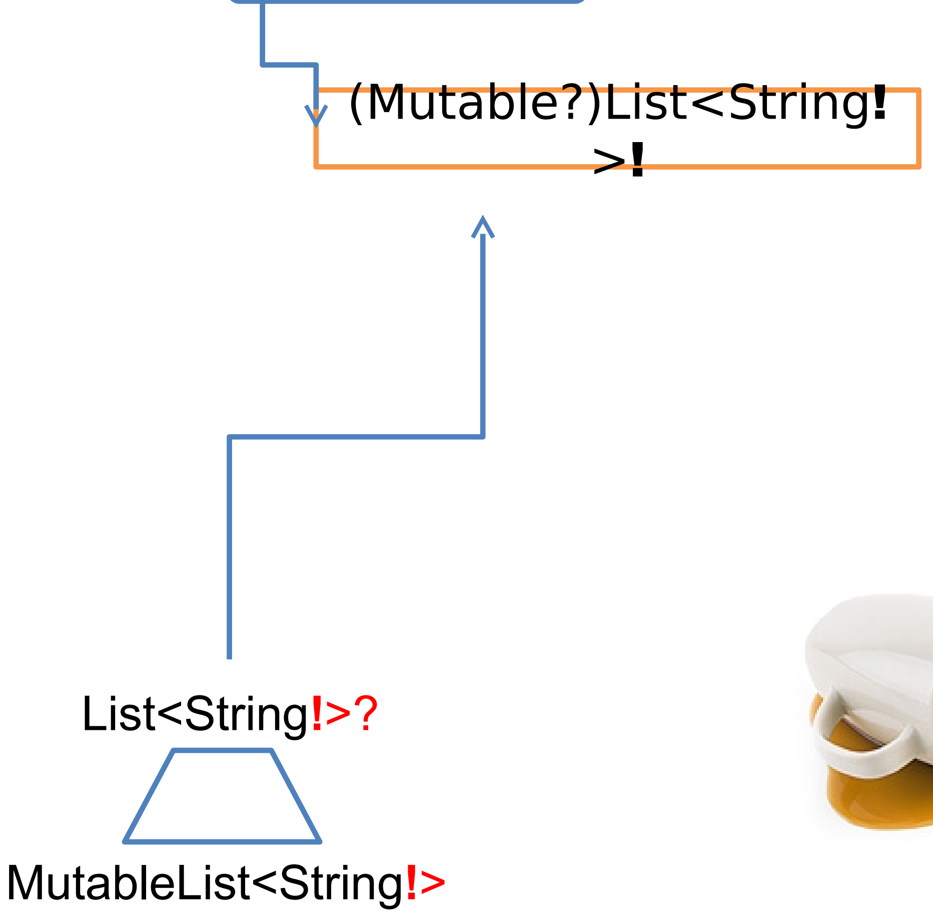


Bar<String**!**>**!**

Flexible Type

Java: Flexible Types

```
public class JavaClass {
    public String foo(List<String> l) {…}
}
```

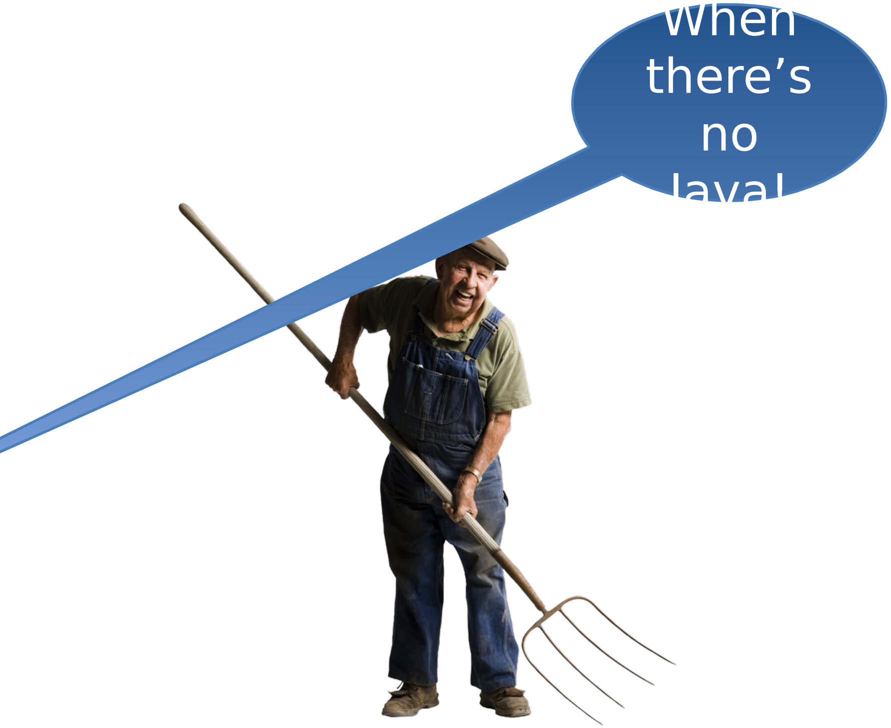


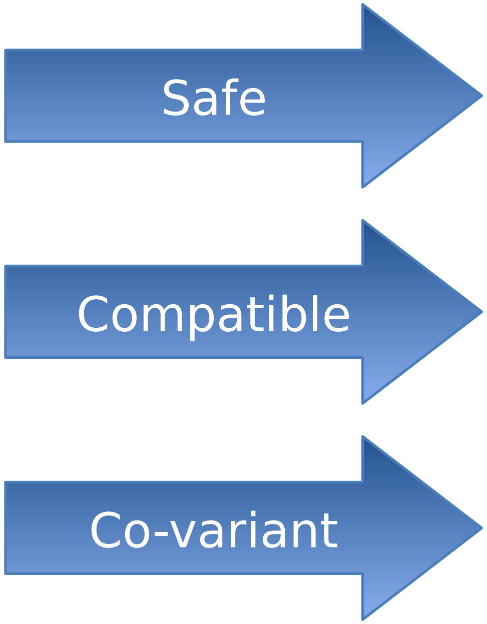
(Mutable?)List<String!>!

List<String!>?

MutableList<String!>

Kotlin Collections



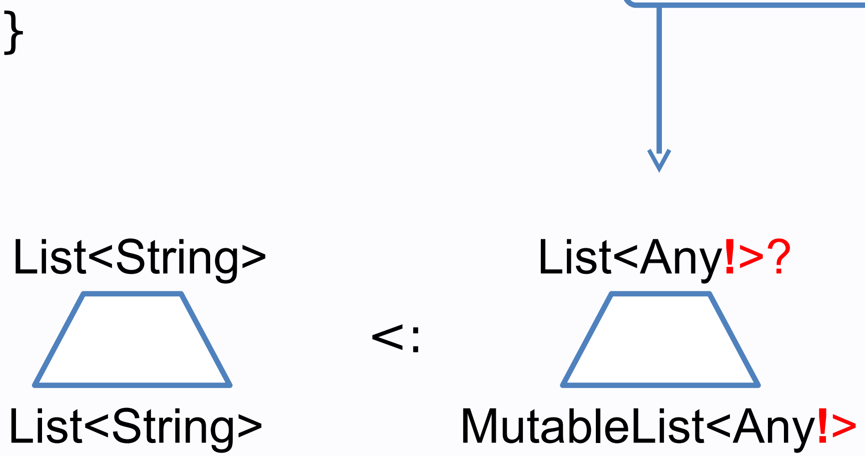

Safe

Compatible

Co-variant

And Then It Gets Bad

```
public class JavaClass {
    public String foo(List<Object> l) {…}
}
```



List<String>

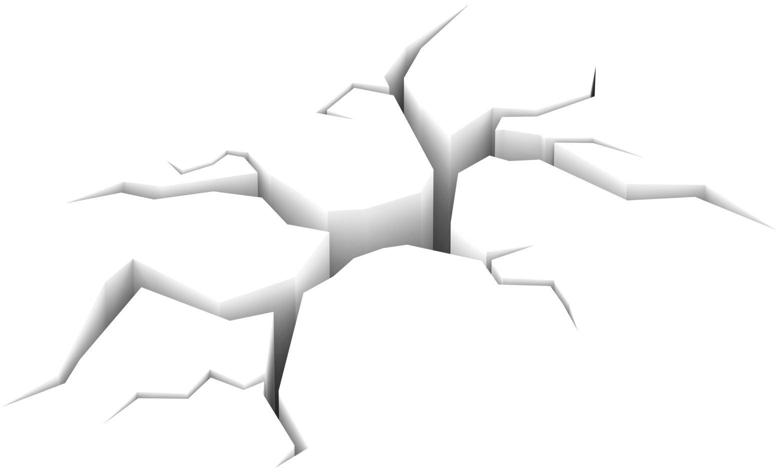
List<String>

<:

List<Any!>?

MutableList<Any!>

```
val strs = listOf(“abc”, “def”)
JavaClass.foo(strs) // ☹
```

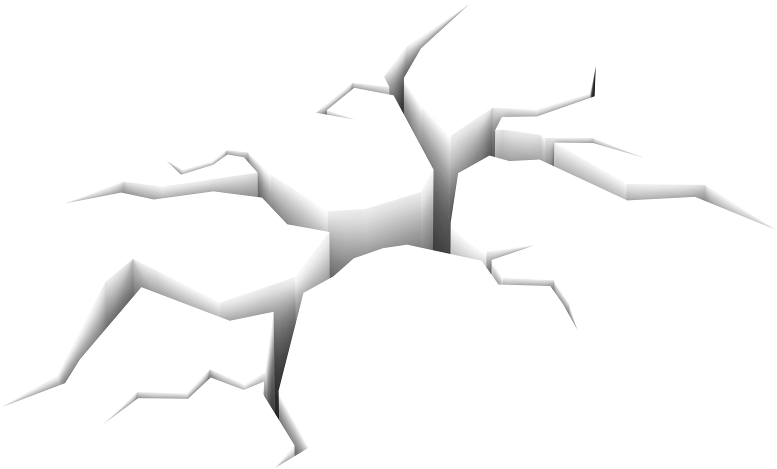


* http://www.psdgraphics.com/file/cracked-floor-background.jpg

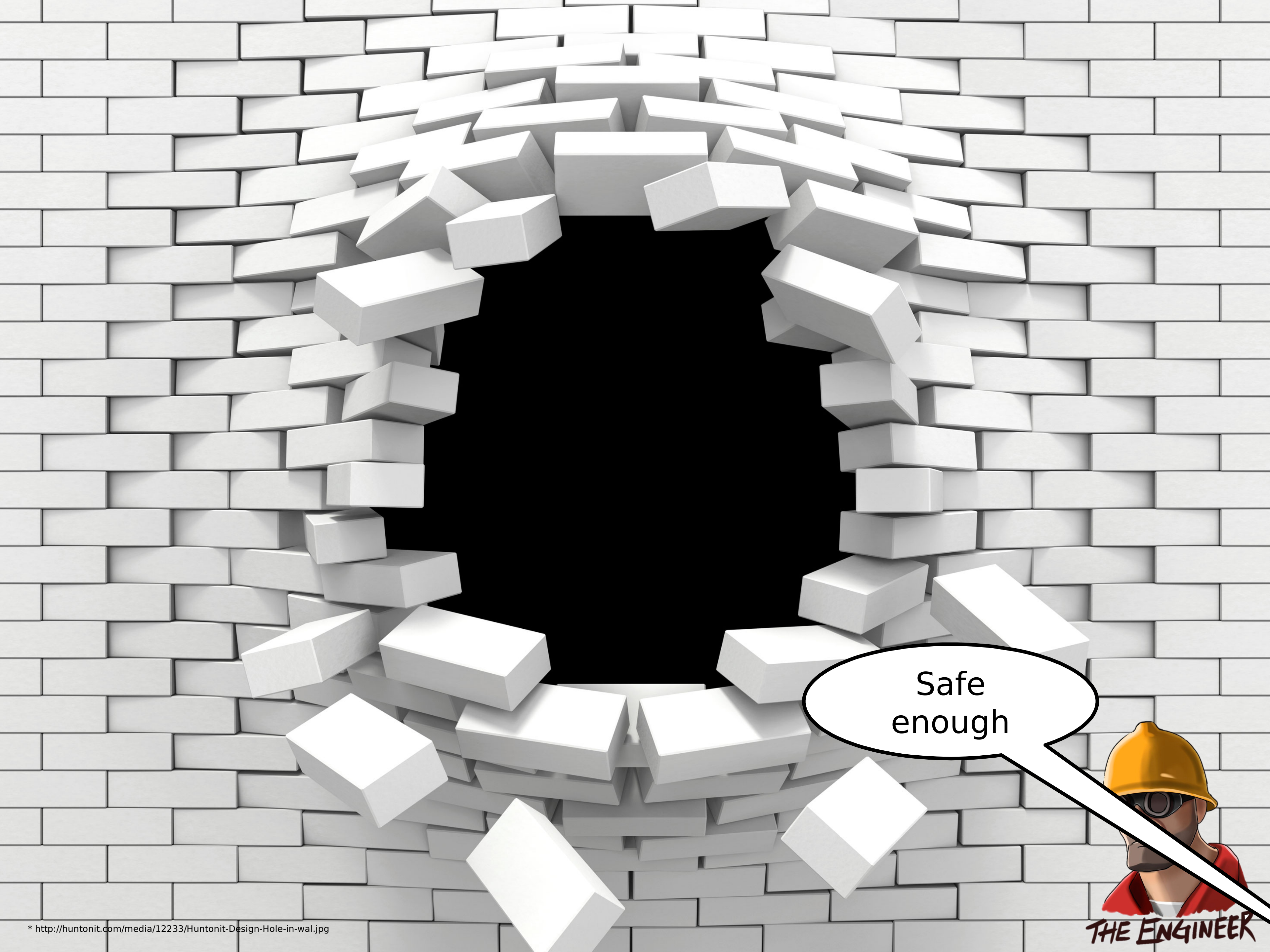
It’s Inevitable ☹

- kotlin.List is covariant
- kotlin.List is not final
- We enable intended use of normal APIs
  - kotlin.List<Any> is assignable to java.util.List<Object>

```
val strs = listOf(“abc”, “def”)
val anys: List<Any> = strs

JavaClass.foo(anys) // ☹
```



* http://www.psdgraphics.com/file/cracked-floor-background.jpg



* http://huntonit.com/media/12233/Huntonit-Design-Hole-in-wal.jpg

Ad Hoc Check

(that cures nothing, but helps a lot)

- Whenever we see a call `javaMethod(kotlinCollection)`
  - After normal subtype cl
  - Perform an extra check

```
val strs = listOf(“abc”, …)
JavaClass.foo(strs) // ☺

val anys: List<Any> = strs
JavaClass.foo(anys) // ☹
```

* http://digitalsophisticate.com/wp-content/uploads/2013/09/Duct-Tape-2.jpg

Runtime Checks
(and their absence)

- `MutableList <- (Mutable)List`
  - Fails if the value is **not mutable**
  - No checks for the list contents
- Where “mutable” means
  - All Java collection implementations
    - including **unmodifiable*()** ☹
  - Kotlin classes that extend **Mutable***

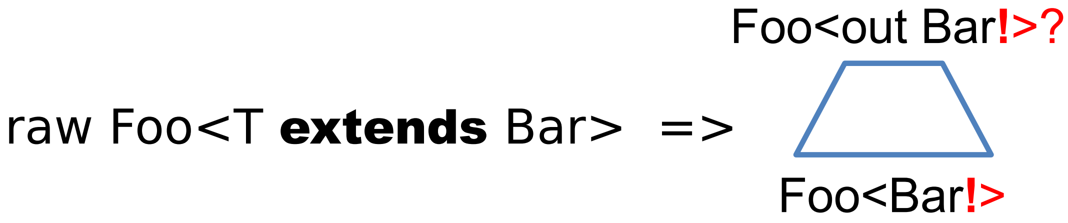
* http://img06.deviantart.net/1744/i/2013/082/5/b/there_s_a_hole_in_the_fence_where_i_used_to_see__by_tricyclemishap-d5z1u3w.jp

Don't EAT Raw

* http://i.ytimg.com/vi/K92S_n79ETk/maxresdefault.jpg

Raw Types

- Still occur in real world
  - More often than I’d like
  - **class** Rec<T **extends** Rec> { … }
- Important for binding overrides



raw Foo<T **extends** Bar> =>

Foo<out Bar!>?

Foo<Bar!>

- Can’t assign raw Foo to Foo<T>, but it’s OK
  - Explicit (uncheced) cast required

Summary (I)

- Nullable types
  - Same as in Java
  - Enhanced by (optional) annotations
  - Some runtime checks
- Arrays
  - Same as in Java
- Collections
  - Worse than Java
  - Hacky ad hoc check (not a cure, but it helps)
  - Enhanced by (optional) annotations

Summary (II)

More safety than Java

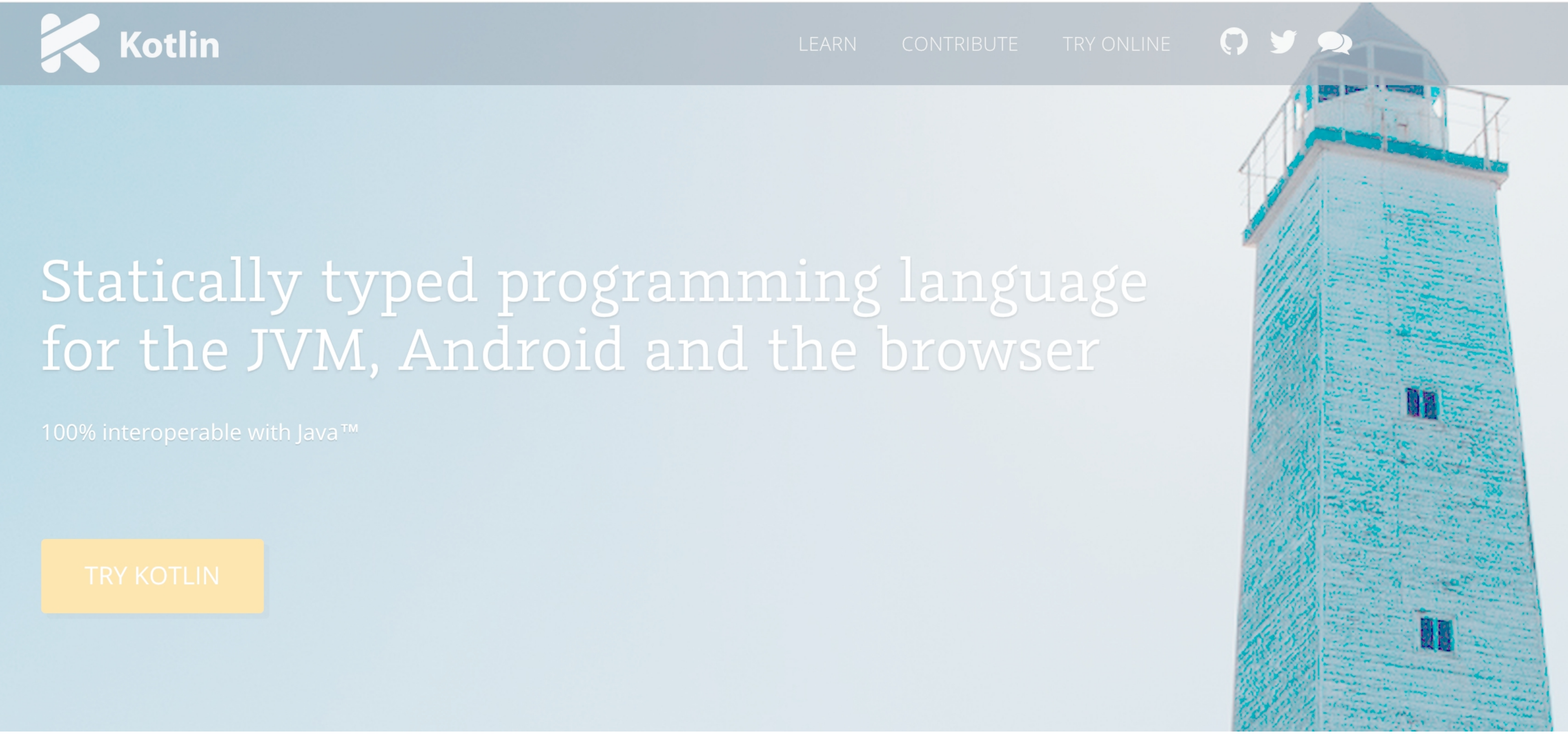
+

Seamless interop

=

“gradual” types that are **unsound**

(i.e. less safety, but **not so often**)



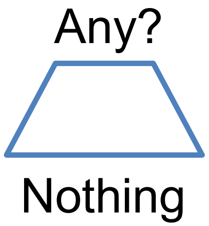
Statically typed programming language for the JVM, Android and the browser

100% interoperable with Java™

http://kotlinlang.org

Expressing **dynamic** types
(for JS)

dynamic =>


Any?
Nothing