Generics

Generic interfaces and classes

```
interface List<E> {
    fun get(index: Int): E
}
```

```
fun foo(ints: List<Int>) { … }

fun bar(strings: List<String>) { … }
```

Generic functions

Generic functions

type parameter

```
fun <T> List<T>.filter(predicate: (T) -> Boolean): List<T> {
    val destination = ArrayList<T>()
    for (element in this) {
        if (predicate(element)) destination.add(element)
    }
    return destination
}
```

Generic functions

```
fun <T> List<T>.filter(predicate: (T) -> Boolean): List<T>
```

```
fun use1(ints: List<Int>) {
    ints.filter { it > 0 }
}

fun use2(strings: List<String>) {
    strings.filter { it.isNotEmpty() }
}
```

Generic functions

```
fun <T> List<T>.filter(predicate: (T) -> Boolean): List<T>
```

```
fun use3(ints: List<Int?>) {
    ints.filter { it != null && it > 0 }
}

fun use4(strings: List<String?>) {
    strings.filter { !it.isNullOrEmpty() }
}
```

Nullable generic argument

```
fun <T> List<T>.firstOrNull(): T?
```

```
val ints = listOf(1, 2, 3)
val i: Int? = ints.firstOrNull()                  // 1

val j: Int? = listOf<Int>().firstOrNull()    // null

val k: Int? = listOf(null, 1).firstOrNull() // null
```

Can `element` be nullable in the example below?

```
fun <T> foo(list: List<T>) {
    for (element in list) {

    }
}
```

1. yes
2. no

Can `element` be nullable in the example below?

```
fun <T> foo(list: List<T>) {
    for (element in list) {

    }
}


foo(listOf(1, null))
```

1. yes
2. no

Non-nullable upper bound

```
fun <T : Any> foo(list: List<T>) {
    for (element in list) {

    }
}
```

```
foo(listOf(1, null))
```

Error: Type parameter bound for `T` is not satisfied:
inferred type `Int?` is not a subtype of `Any`

Non-nullable upper bound

```
fun <T : Any> List<T?>.filterNotNull(): List<T> { … }
```

```
val list: List<Int> = listOf(1, null).filterNotNull()
```

Non-nullable upper bound

```
fun <T : Any> List<T?>.filterNotNull(): List<T> { … }
```

```
val list: List<Int> = listOf(1, null).filterNotNull()
```

Non-nullable upper bound

```
fun <T : Any> List<T?>.filterNotNull(): List<T> { … }
```

```
val list: List<Int> = listOf(1, null).filterNotNull()
```

Type parameter constraints

```
fun <T : Number> oneHalf(value: T): Double {
    return value.toDouble() / 2.0
}
```

```
oneHalf(13)   // 6.5
```

Nullable upper bound

```
fun <T : Number?> oneHalf(value: T): Double? {
    if (value == null) return null
    return value.toDouble() / 2.0
}
```

```
oneHalf(13)   // 6.5
oneHalf(null) // null
```

`Comparable` upper bound

```
fun <T : Comparable<T>> max(first: T, second: T): T {
    return if (first > second) first else second
}
```

```
max(1, 3)  // 3
```

Multiple constraints for a type parameter

```
fun <T> ensureTrailingPeriod(seq: T)
        where T : CharSequence, T : Appendable {
    if (!seq.endsWith('.')) {
        seq.append('.')
    }
}
```

```
val helloWorld = StringBuilder("Hello, World")
ensureTrailingPeriod(helloWorld)
println(helloWorld)  // Hello, World.
```

Same JVM signature

Error: Platform declaration clash:
The following declarations have
the same JVM signature
`average(Ljava/util/List;)D`

```
fun List<Int>.average(): Double { … }

fun List<Double>.average(): Double { … }
```

Applying which annotation helps to solve this problem?

```
fun List<Int>.average(): Double { … }

@???
fun List<Double>.average(): Double { … }
```

Applying which annotation helps to solve this problem?

```
fun List<Int>.average(): Double { … }

@JvmName("averageOfDouble")
fun List<Double>.average(): Double { … }
```

By which name you can call the second function from Java?

```
fun List<Int>.average(): Double { … }

@JvmName("averageOfDouble")
fun List<Double>.average(): Double { … }
```

1. average
2. averageOfDouble

By which name you can call the second function from Java?

```
fun List<Int>.average(): Double { … }

@JvmName("averageOfDouble")
fun List<Double>.average(): Double { … }
```

1. average
2. averageOfDouble

Later

```
interface List<out E> {
    fun get(index: Int): E
}
```

```
inline fun <reified R> List<*>.filterIsInstance(): List<R>
```