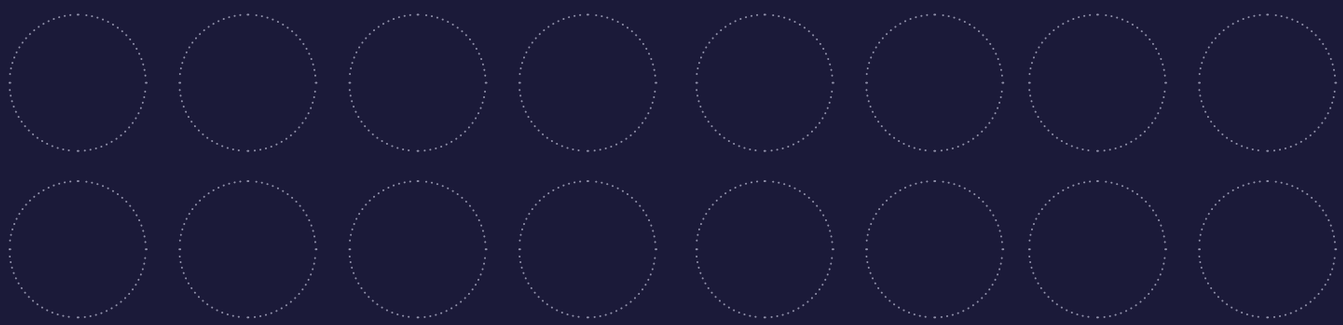
Kotlin

Future Features

1 Multi-catch



Sometimes we need to handle more than one exception in a single catch block:

```
try {
    // some code
} catch (e: ExceptionA|ExceptionB) {
    log(e);
    throw e;
}
```

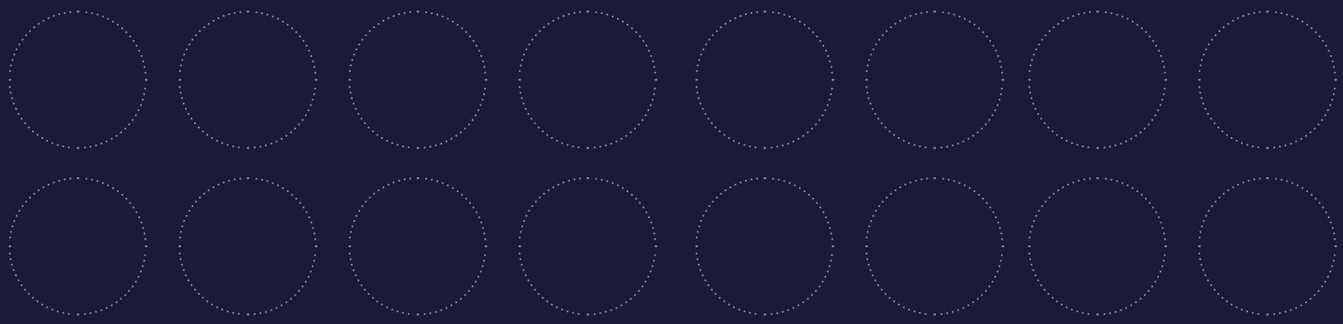
Note that this is not proposing general union types:
TypeA|TypeB will only be allowed in the context of a catch variable.

Kotlin

Future Features

2

Overloadable operators | and &



Kotlin has || and && for booleans. These operators can not be overloaded, because of their short-circuit semantics: right-hand side of ||, for example, won't be evaluated is the left-hand side is already true.

Overloadable operators | and &, with no short-circuit semantics (both sides will be evaluated regardless) could be useful for DSLs as well as for traditional bitwise operations on integers.

```
fun test() {
    task1 | task2
}class Task {
    operator fun or(other: Task) = /* pipe output of this into other */
}
fun test() {
    task1 | task2
}
```

Kotlin

Future Features

3

Short notation for enum constants

Referring to enum constants with EnumName.CONST_NAME often looks verbose. The compiler could get smarter and infer the relevant enum's name from context.

Example:

```
fun test(c: Color) = when (c) {
    RED -> { ... }
    GREEN -> { ... }
    BLUE -> { ... }
}
```

Same for assignments:

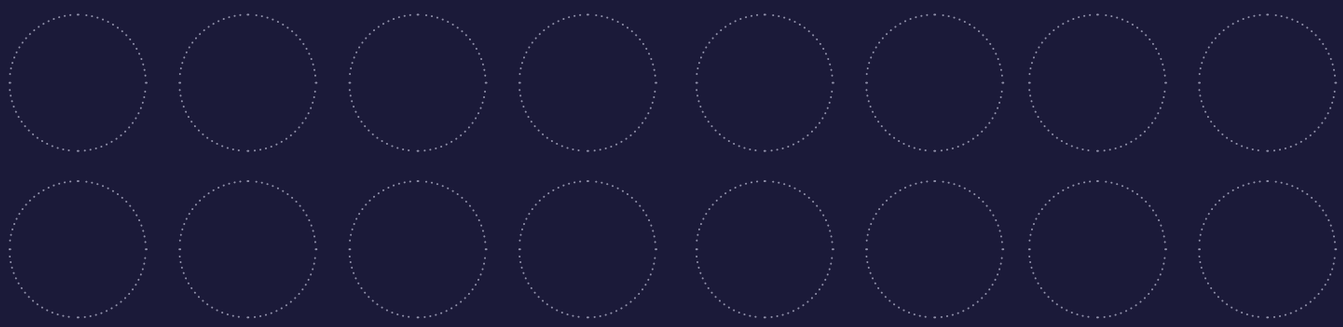
```
widget.color = RED
```

Kotlin

Future Features

4

Private members accessible from tests



We test a lot more than we expose as an API. Making things internal or package-private just for the sake of testing them can cause confusion and leaky abstractions: clients inside the module/package can still call and abuse these methods, so this technique is a problem for a good API.

Production code:

```
class ProductionClass {
    private var privateField = ...
    public fun method() { ... }
}
```

Tests:

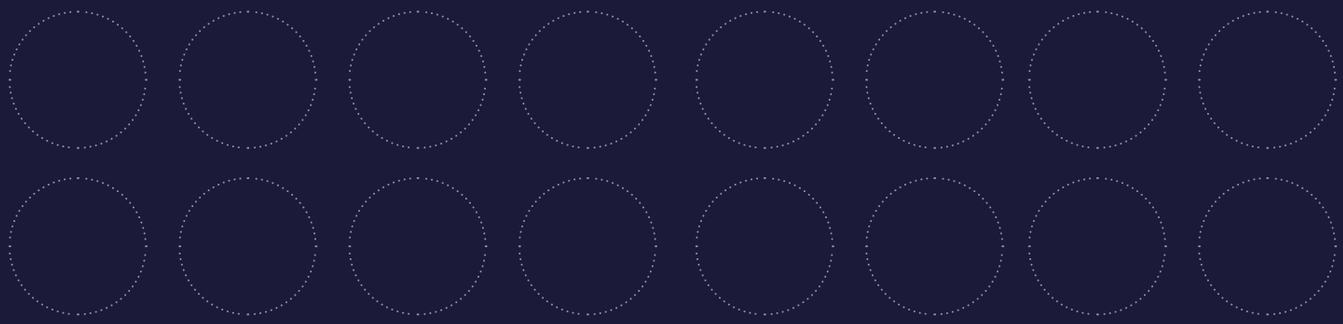
```
@Test
fun barFoo() {
    val p = ProductionClass()
    p.method()
    assertTrue("...", p.
privateField) // Allowed only in
tests!
}
```

Kotlin

Future Features

5

Support package-private visibility



In Kotlin packages are namespaces and not visibility scopes. Modules, on the other hand, are visibility scopes and not namespaces. So, there's **internal** visibility (visible inside a module), but no **package-private** visibility.

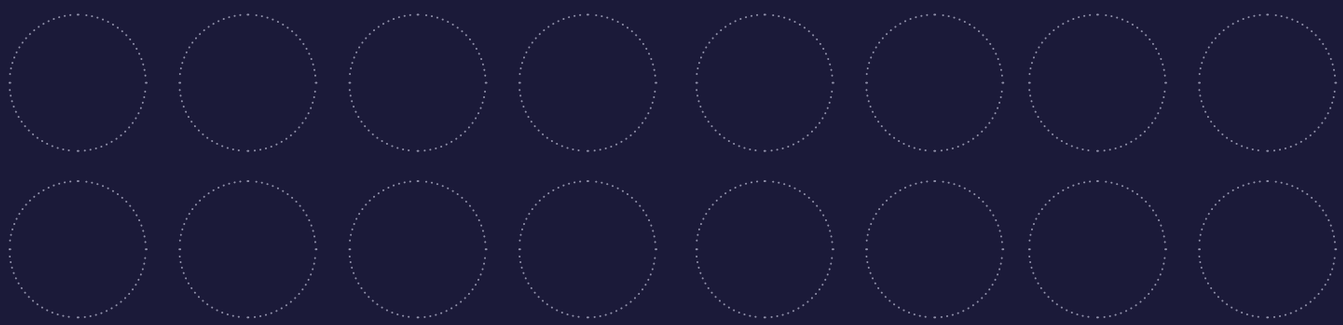
Some users report issues in mixed Kotlin/Java projects where they want Kotlin to have **package-private**, just like Java.

Do you want this too?

Kotlin

Future Features

6 Collection literals



Today, we create collections in Kotlin by calling library functions, such as `listOf(1, 2, 3)` or `mapOf("foo" to "bar", "baz" to "roo")`. We could make it more convenient by adding specialized syntax, e.g. `[1, 2, 3]` for a list or an array, and something like `["foo" = "bar", "baz" = "roo"]`.

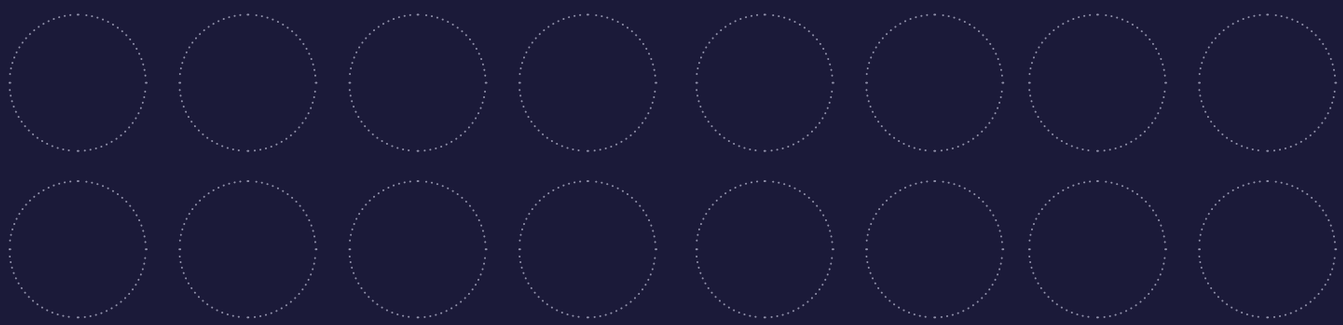
All syntax is purely provisional at this point.

Particular types of collections created may be determined by libraries/operator conventions.Annotations can benefit from this through shortened syntax for arrays: `@RequestMapping(path = ["/"])`.

Kotlin

Future Features

7 Collection comprehensions



Some languages support short notation for constructing collections (that basically combines maps and filters in a very terse syntax). Examples would be list comprehensions in Python/ Haskel, and LINQ in C#, as well as Scala's `for/yield`.

Kotlin could also have some syntax like

```
[x.name for x in employees if x.division == Production]
```

instead of

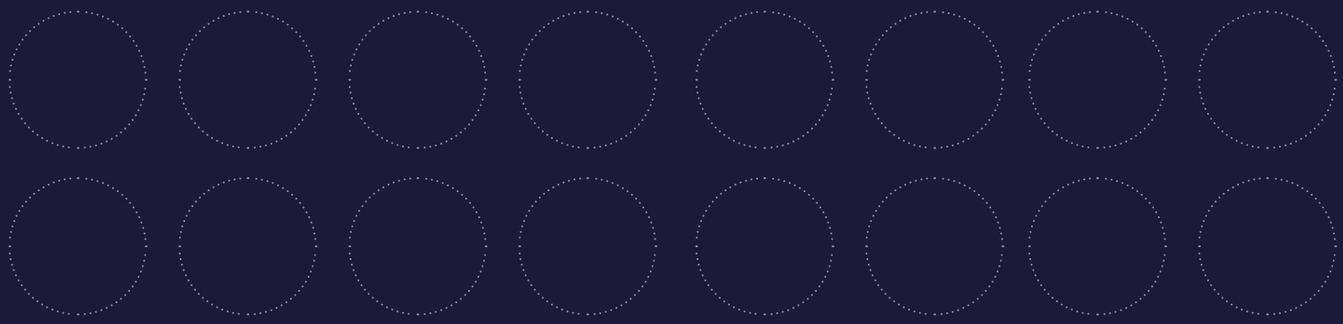
```
employees.filter {it.division == Production}.map {it.name}
```

Kotlin

Future Features

8

Slices for lists and arrays



Slices are popular in numeric computations, where

`data[a:b]` means sub-list from a to b-1

`data[a:]` means sub-list from a to the end

`data[a:b:c]` means sub-list from a to b-1 with step c

Kotlin

Future Features

9

Inline classes/ Value classes

Many of us want to be able store aggregate data by value, not only by reference, but the VM's Kotlin runs on at the moment (JVM/Android/JS) do not allow this, at least until value types are implemented for the JVM (Project Valhalla).

So, instead we can use some compiler magic to support value classes *with only one (immutable) field*. For example:

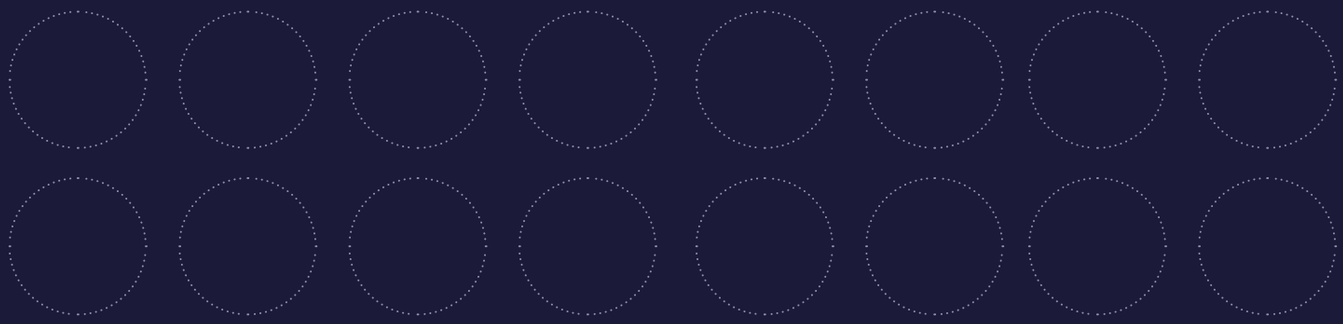
```
inline class MyDate(private val data: Long) {
    val month: Int get() = ...
    val year: Int get() = ...

    fun shiftBy(days: Int) = ...
}
```

This could be used as a normal class, but stored as a Long, with no wrapper object allocated. Such classes could be useful to represent units of measurement, unsigned arithmetic and custom strings — when a piece of data (like a hash or a user ID) is represented by a string, but we don't want this to be freely mixed with other strings.

Kotlin

Future Features

10 Format strings



Formatting directives for string interpolations could be a part of the language syntax (unlike current String.format(...)). Syntax is only provisional, but you can get the idea:

`“${.2d: 123.339}”`
is the same as `String.format(“%.2d”, 123.339)`

`“${e: x / y}”`
is the same as `String.format(“%e”, x / y)`

The exact requirements, operator conventions and syntactic forms are yet to be designed.

Kotlin

Future Features

11

Optional (trailing) commas

Many comma-separated lists are hard to re-order because the last item is different from all the others: it has no comma after it.

Trailing commas

We propose to allow a comma after last elements in such lists:

```
data class Foo(val a: Int, val
b: String, )
fun foo<X, Y,>(a: Int, b: String,
) { ... }
val foo = foo<X, Y, >(a, b, )
```

Commas before a new line

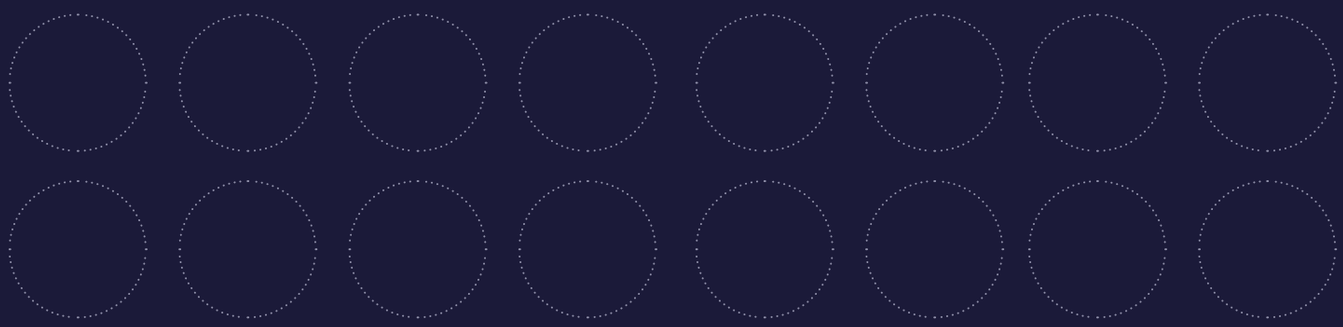
We propose to allow no comma after any element of a list if it's followed by a new line:

```
fun foo(
        a: Int            // no comma
        b: String
) { ... }data class Foo(
        val a: Int        // no comma
        val b: String
)
fun foo(
        a: Int            // no comma
        b: String
) { ... }
```

Kotlin

Future Features

12 Unsigned arithmetic



Support types such as `UInt, ULong, UByte, UShort` representing unsigned integers. This way `0xFFFFFFFF` (currently doesn't fit into Int) will become a legitimate value of type `UInt`.

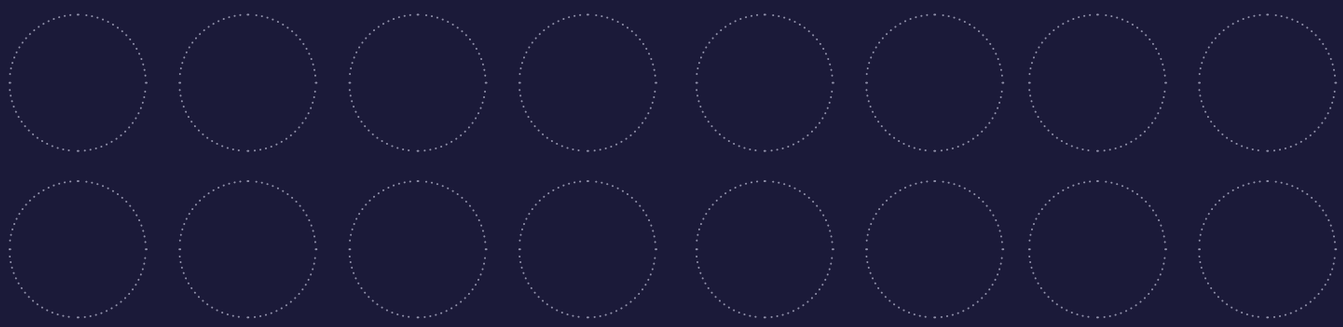
Unsigned numbers should support all the common operations (`+, -, *, /, %, >, <, >=, <=, and, or, xor, inv, shl, shr`), but treat the most significant bit as a regular one, not a sign. This means, for example that an `UByte` value `0xFF` is greater than `0`, not less as with signed Byte

Kotlin

Future Features

13

SAM conversions for Kotlin interfaces



Interfaces that have a Single Abstract Method (SAM-interfaces) can be naturally implemented by lambdas. In fact, this is how lambdas work in Java 8: when it's assigned to an SAMinterface, it implements it. This is called a SAM conversion.

Kotlin has SAM conversions for Java interfaces.
But if this interface is defined in Kotlin, we can't assign a lambda directly to it:w

```
interface Action<T> {
    fun run(data: T)
}
```

Here we propose to allow assigning lambdas to SAM interfaces:

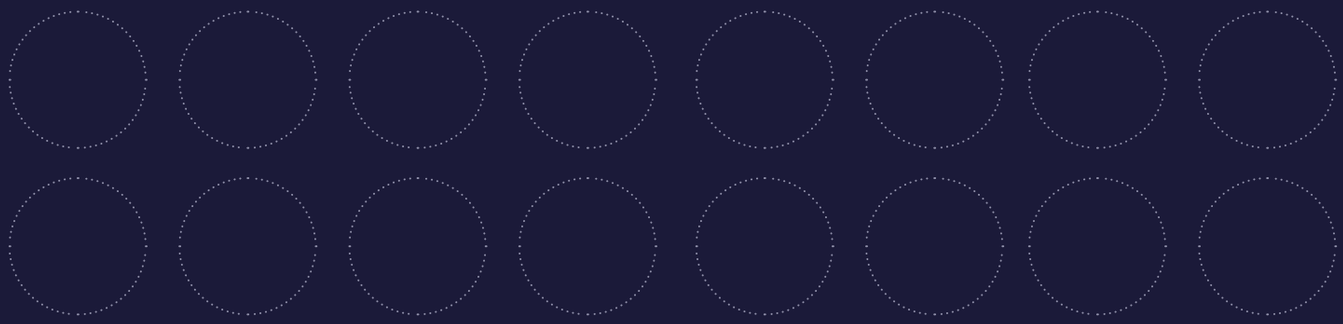
```
val action: Action<T> = { data -> log("data: $data") }
```

Kotlin

Future Features

14

Annotations for static analyses



Many interesting properties of programs can be verified by a static analyzer (built into the compiler or provided as a plugin). We could support annotations to denote such properties and issue errors if they are not satisfied:

`@Pure` for functions with no side-effects, constructors and property accessors

`@MustUseReturnValue` for functions whose return value must never be ignored (like Futures,error codes, etc)

`@MustCatch` for opt-in checked exceptions

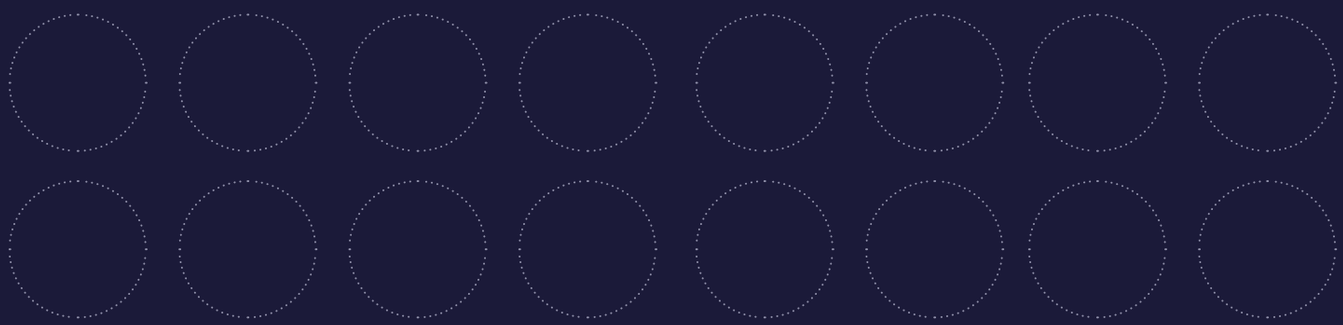
`@RunUnconditionally` for inline lambdas that are run unconditionally and thus work as simple code blocks (can allow smart casts, val initialization and other such things)

`@SafeInitialization` to make sure that virtual calls in constructors do not break any logic

Kotlin

Future Features

15 Destructuring assignments



Kotlin already has destructuring declarations:

```
val (name, address) = findPerson(...)
```

Some users request destructuring assignments, i.e. assign to multiple previously declared `var`**'s:**

```
var name = ...
...
var address = ...
...
(name, address) = findPerson(...)
```

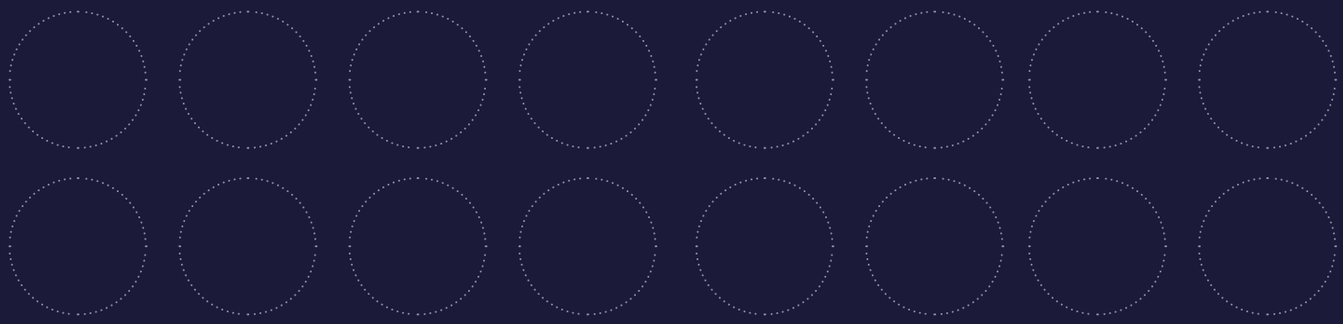
Do you need this feature?

Kotlin

Future Features

16

Use `invokedynamic` to compile Kotlin



When generating Java 8 byte codes, we can compile Kotlin lambdas using `invokedynamic` (the same way that Java 8 does this). Compared to the current compilation strategy (anonymous inner classes, used for Java 6 byte code) it means fewer class files and slightly fewer bytes in the output.

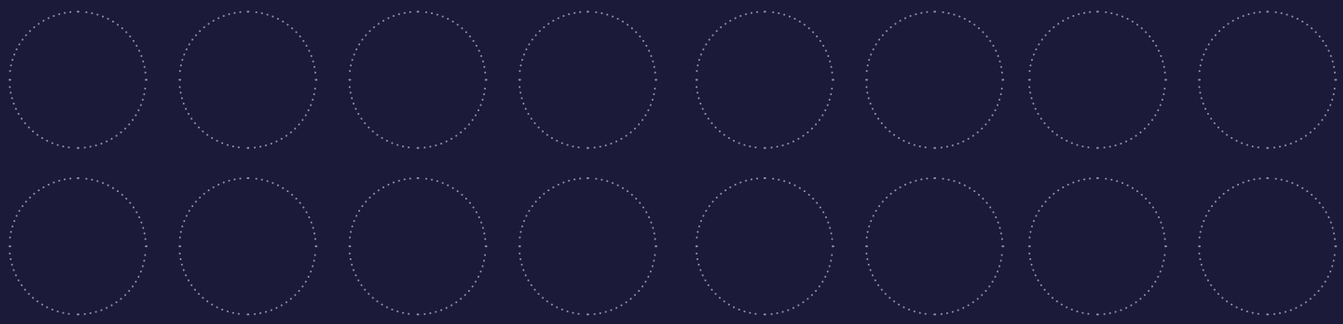
NOTE: It will not reduce method counts for Android .APK's, because Android doesn't have invokedynamic and rewrites it back into inner classes.

Kotlin

Future Features

17

Static members for Kotlin classes



Kotlin doesn't have static members in classes or interfaces. Instead, it has companion objects. This concept causes some issues with Java interop and sometimes causes boilerplate, and we are wondering if these issues are serious enough to eventually introduce true static members to Kotlin:

```
class Example {
    fun instanceMember() { ... }

    static fun staticMember() { ... }
}
```

What do you think?

Kotlin

Future Features

18

Truly immutable data

Kotlin has immutable variables (**val**'s), but no language mechanisms that would guarantee true "deep" immutability of the state. If a val references a mutable object, it's contents can be modified:

```
val myVal = arrayListOf(1)
myVal.add(2) // mutation
```

Read-only interfaces for collections help somewhat, but we'd need something like **readonly**/**immutable** modifier for *all types* to ensure true immutability. Syntax is purely provisional:

```
class Foo(var v: Int)

immutable class Bar(val foo: Foo) // error: mutable reference from immutable
class
```

Kotlin

Future Features

19 Subject variable in when

When examining a value of a complex expression, it may be useful to refer to it as a variable.

Example

```
when (val x = some.complex(expression)) {
    A -> log(x)
    B -> blog(x)
    C -> balrog(x)
}
```

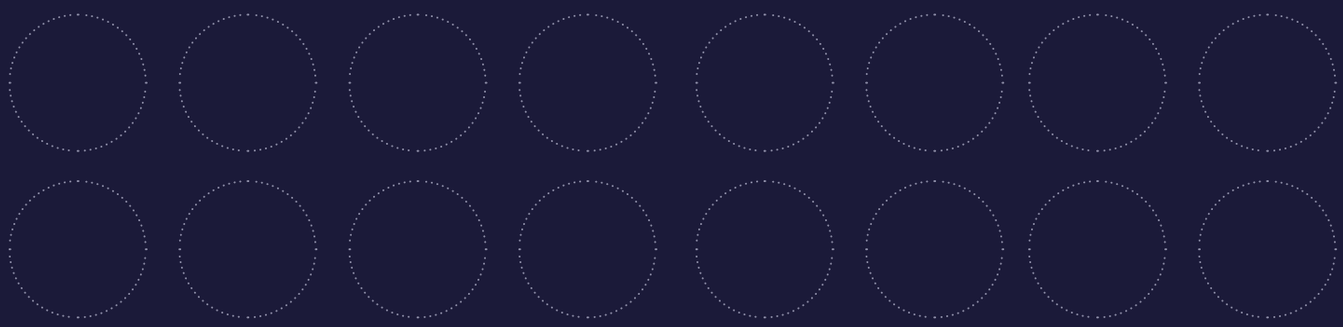
A variation of this proposal is to use “it” as an implicitly defined name for any when’s subject, but we feel that this would interact badly with “it” in lambdas.

Kotlin

Future Features

20

Vararg-like treatment of data classes



We have varargs for passing arrays to functions without explicitly creating them. We could have the same for data classes:

```
// Usage:
doSomething(name = ”John Doe”, age = ”25”)data class Person(val name: String,
val age: Int)
fun doSomething(dataarg p: Person) { ... )
// Usage:
doSomething(name = “John Doe”, age = “25”)
```