The power of `inline`

Regular non-inlined lambdas

```
fun myRun(f: () -> Unit) = f()

fun main(args: Array<String>) {
    val name = "Kotlin"
    myRun { println("Hi, $name!") }
}
```

class ExampleKt$main$1

Regular non-inlined lambdas

```
fun myRun(f: () -> Unit) = f()

fun main(args: Array<String>) {
    val name = "Kotlin"

    // brings performance overhead
    myRun { println("Hi, $name!") }

    // in comparison to:
    println("Hi, $name!")
}
```

`inline` function

compiler substitutes a body of the function
instead of calling it

```kotlin
inline fun <R> run(block: () -> R): R = block()
```

```kotlin
val name = "Kotlin"
run { println("Hi, $name!") }
```

Generated code (in the bytecode):

```kotlin
val name = "Kotlin"
println("Hi, $name!")
```

inlined code of lambda body

Regular non-inlined lambdas

```
fun myRun(f: () -> Unit) { f() }

fun main(args: Array<String>) {
    val name = "Kotlin"

    // brings performance overhead
    myRun { println("Hi, $name!") }

    // in comparison to:
    println("Hi, $name!")
}
```

Inlined lambdas

```
inline fun run(f: () -> Unit) { f() }

fun main(args: Array<String>) {
    val name = "Kotlin"

    // NO performance overhead
    run { println("Hi, $name!") }

    // in comparison to:
    println("Hi, $name!")
}
```

```
inline fun <T> T.takeUnless(predicate: (T) -> Boolean): T? =
    if (!predicate(this)) this else null
```

```
fun foo(number: Int) {
    val result = number.takeUnless { it > 10 }
    ...
}
```

Generated code (in the bytecode):

inlined code of lambda body

```
fun foo(number: Int) {
    val result = if (!(number > 10)) number else null
    ...
}
```

```
inline fun <T> synchronized(lock: Lock, action: () -> T): T {
    lock.lock()
    try {
        return action()
    } finally {
        lock.unlock()
    }
}
```

```
fun foo(lock: Lock) {
    synchronized(lock) {
        println("Action")
    }
}
```

Generated code (in the bytecode):

```
fun foo(lock: Lock) {
    lock.lock()
    try {
        println("Action")
    } finally {
        lock.unlock()
    }
}
```

inlined code of lambda body

withLock function

```
val l: Lock = ...
l.withLock {
    // access the resource protected by this lock
}
```

```
inline fun <T> Lock.withLock(action: () -> T): T {
    lock()
    try {
        return action()
    } finally {
        unlock()
    }
}
```

Resource management: `use` function

```java
/* Java */
static String readFirstLineFromFile(String path)
        throws IOException {
    try (BufferedReader br =
                new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

```kotlin
fun readFirstLineFromFile(path: String): String {
    BufferedReader(FileReader(path)).use { br ->
        return br.readLine()
    }
}
```

No performance overhead when you use

`run, let, takeIf, takeUnless, repeat, withLock, use`

No anonymous class or extra objects are created for lambda under the hood

What code will be generated for the `filter` function?

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

```
fun filterNonZero(list: List<Int>) = list.filter { it != 0 }
```

Will the `filter` function be inlined in the bytecode if you call it from Java?

```
public static void foo(List<Integer> list) {
    List<Integer> positive =
        CollectionsKt.filter(list, element -> element > 0);
}
```

1. yes
2. no

?

Will the `filter` function be inlined in the bytecode if you call it from Java?

```
public static void foo(List<Integer> list) {
    List<Integer> positive =
        CollectionsKt.filter(list, element -> element > 0);
}
```

1. yes
2. no

@InlineOnly

Specifies that this function should not be called directly without inlining

```
@kotlin.internal.InlineOnly
public inline fun <R> run(block: () -> R): R = block()
```

Use `inline` with care