



Learning Kotlin: a trial-and-error approach

Evgeny Trushin

- [linkedin.com/in/evgeny-trushin](https://www.linkedin.com/in/evgeny-trushin)
- [github.com/evgeny-trushin/Kotlin trial-and-error learning](https://github.com/evgeny-trushin/Kotlin-trial-and-error-learning)



Languages I have used in production environments:


- *MEX for Maximus BBS, 1998*
- *Borland Pascal, 2000*
- *C, 2001*
- *HTML/CSS, 2003*
- *PHP/SQL, 2006*
- *JavaScript, 2007*
- *Java, 2012*
- *Objective-C, 2013*
- *Kotlin, 2017*

I have learned plenty of languages and made at least 10,000 coding mistakes in these 20 years.

All the same, I know that the ability to solve problems while coding is the primary skill that separates proficient programming from not being able to program at all.

The goal of this speech is to present you common bug fixing strategies and coding errors which you may come across while mastering the Kotlin language.





When someone says:
"You should use Kotlin instead of Java"

- *Sounds good, doesn't work*

The best strategies that
I can recommend to deal
with Kotlin errors:

1. Ask Google with this context filter:
"site:stackoverflow.com Kotlin" + error

2. Search in the Kotlin documentation.
"site:https://kotlinlang.org/docs/reference" + error

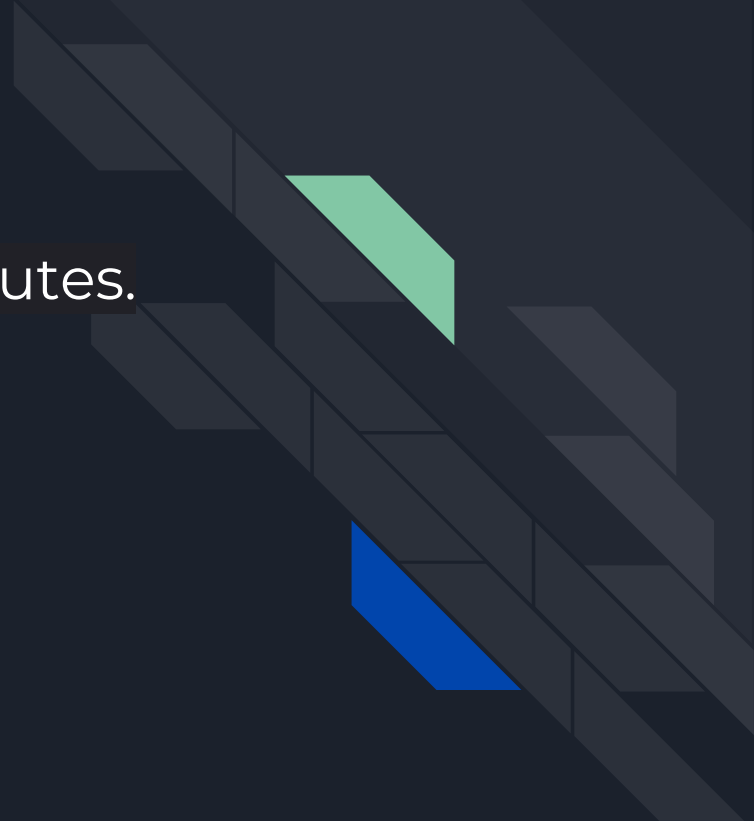
3. Look up the solution in a book.
Google a book with a context filter filetype:pdf Kotlin


The best strategies that
I can recommend to deal
with Kotlin errors:

4. Ask a colleague/mentor/friend.

5. Try to solve it on your own in 30 minutes.

6. Rewrite your code differently and
repeat these strategies.





Let's try to apply some
of those strategies.



Variables...



Catch an error: unresolved reference

```
var Int: b = 2  
println("b equals " + b)
```



Catch an error: unresolved reference

```
var Int: b = 2  
println("b equals " + b)
```

```
var b: Int = 2  
println("b equals " + b)
```



Catch an error: **val** cannot be **reassigned**

```
val c = 2  
c = 3  
println("c equals $c")
```



Catch an error: **val** cannot be **reassigned**

```
val c = 2
```

```
c = 3
```

```
println("c equals $c")
```

```
var c = 2
```

```
c = 3
```

```
println("c equals $c")
```



Catch an error: the **integer** literal does not **conform** to the expected type **String**

```
var a = "Test"  
a = 10  
println("a equals $a")
```



Catch an error: the **integer** literal does not **conform** to the expected type **String**

```
var a = "Test"  
a = 10  
println("a equals $a")
```

```
var a = "Test"  
a = "10"  
println("a equals $a")
```



Catch an error: **overload** resolution **ambiguity**

```
var c: Int = 2  
val c = 2  
println("c equals $c")
```



Catch an error: **overload** resolution **ambiguity**

```
var c: Int = 2  
val c = 2  
println("c equals $c")
```

```
var c: Int = 2  
val cSecondary = 2  
println("c equals $c")
```




Catch an error: property getter or setter expected

```
val int a = 1  
println("a equals $a")
```



Catch an error: property getter or setter expected

```
val int a = 1  
println("a equals $a")
```

```
val a = 1  
println("a equals $a")
```



Catch an error: captured member values
initialization is forbidden due to possible
reassignment

```
val c  
c = 3  
println("c equals $c")
```




Catch an error: captured member values
initialization is forbidden due to possible
reassignment

```
val c  
c = 3  
println("c equals $c")
```

```
val c = 3  
println("c equals $c")
```




Nullable types...



Catch an error: only safe **(?.)** or non-null asserted **(!!.)** calls are allowed on a **nullable receiver**

```
class Test(var flag: Boolean)

var a: Test? = null
a = Test(true)
a.flag = false
println("a.isFlag equals ${a.flag}")
```



Catch an error: only safe **(?.)** or non-null asserted **(!!.)** calls are allowed on a **nullable receiver**

```
class Test(var flag: Boolean)
```

```
var a: Test? = null
```

```
a = Test(true)
```

```
a.flag = false
```

```
println("a.isFlag equals ${a.flag}")
```

```
class Test(var flag: Boolean)
```

```
var a: Test? = null
```

```
a = Test(true)
```

```
a?.flag = false
```

```
println("a.isFlag equals ${a?.flag}")
```



Functions...



Catch an error: unresolved reference

```
fun fixme(Int: a) {  
    println("Param a equals " + a)  
}  
fixme(1)
```



Catch an error: unresolved reference

```
fun fixme(Int: a) {  
    println("Param a equals " + a)  
}  
fixme(1)
```

```
fun fixme(a: Int) {  
    println("Param a equals " + a)  
}  
fixme(1)
```



Catch an error: type mismatch: **'return'** is not allowed here

```
fun fixme1(a: Int) = { return a + 1 }  
fun fixme2(a: Int): Int { return a + 1 }  
fun fixme3(a: Int): Int = a + 1  
println(fixme1(1))  
println(fixme2(1))  
println(fixme3(1))
```



Catch an error: type mismatch: '**return**' is not allowed here

```
fun fixme1(a: Int) = { return a + 1 }  
fun fixme2(a: Int): Int { return a + 1 }  
fun fixme3(a: Int): Int = a + 1  
println(fixme1(1))  
println(fixme2(1))  
println(fixme3(1))
```

```
fun fixme1(a: Int) = a + 1  
fun fixme2(a: Int): Int { return a + 1 }  
fun fixme3(a: Int): Int = a + 1  
println(fixme1(1))  
println(fixme2(1))  
println(fixme3(1))
```



Classes...



Catch an error: unresolved reference: **new**

```
class Test(val test: Int)

val a = new Test (1)
println("a equals ${a.test}")
```



Catch an error: unresolved reference: **new**

```
class Test(val test: Int)
```

```
val a = new Test (1)  
println("a equals ${a.test}")
```

```
class Test(val test: Int)
```

```
val a = Test(1)  
println("a equals ${a.test}")
```



Catch an error: unresolved reference: **test**

```
class Test(test: Int)

val a = Test(1)
println("a equals ${a.test}")
```




Catch an error: unresolved reference: **test**

```
class Test(test: Int)
```

```
val a = Test(1)  
println("a equals ${a.test}")
```

```
class Test(val test: Int)
```

```
val a = Test(1)  
println("a equals ${a.test}")
```



Catch an error: unresolved reference: **isFlag**

```
class Test(var flag: Boolean)

val a = Test(true)
println("a.isFlag equals ${a.isFlag}")
```



Catch an error: unresolved reference: **isFlag**

```
class Test(var flag: Boolean)
```

```
val a = Test(true)
```

```
println("a.isFlag equals ${a.isFlag}")
```

```
class Test(var isFlag: Boolean)
```

```
val a = Test(true)
```

```
println("a.isFlag equals ${a.isFlag}")
```



Catch an error: primary constructor call
expected **constructor(val test: Int)**

```
class Test() {  
    constructor(val test: Int)  
}
```

```
val a = Test(1)  
println("a equals ${a.test}")
```




Catch an error: primary constructor call expected **constructor(val test: Int)**

```
class Test() {  
    constructor(val test: Int)  
}
```

```
val a = Test(1)  
println("a equals ${a.test}")
```


```
class Test(val test: Int)  
  
val a = Test(1)  
println("a equals ${a.test}")
```



Catch an error: overload resolution
ambiguity: public constructor **Info(nameA: String)** defined in Fixme.Info

```
data class Info(val nameB: String) {  
    constructor(nameA: String)  
        : this(nameA)  
}
```

```
println(Info(nameA = "nameA").nameB)
```



Catch an error: overload resolution
ambiguity: public constructor **Info(nameA:
String)** defined in Fixme.Info

```
data class Info(val nameB: String) {  
    constructor(nameA: String)  
        : this(nameA)  
}
```

```
println(Info(nameA = "nameA").nameB)
```

```
data class Info(val nameB: String) {  
    constructor(nameA: String)  
        : this(nameB = nameA)  
}
```

```
println(Info(nameA = "nameA").nameB)
```



Arrays...



Catch an error: unresolved reference: **add**

```
val a = intArrayOf(9, 11)
a.add(14)
println("a[0] " + a[0])
println("a[1] " + a[1])
println("a[2] " + a[2])
```



Catch an error: unresolved reference: **add**

```
val a = intArrayOf(9, 11)
a.add(14)
println("a[0] " + a[0])
println("a[1] " + a[1])
println("a[2] " + a[2])
```


```
val a = arrayListOf(9, 11)
a.add(14)
println("a[0] " + a[0])
println("a[1] " + a[1])
println("a[2] " + a[2])
```



Catch an error: unresolved reference: **add**


```
val a = intArrayOf(9, 11)
a.add(14)
println("a[0] " + a[0])
println("a[1] " + a[1])
println("a[2] " + a[2])
```

```
val a = ArrayList<Int>()
a.add(9)
a.add(11)
a.add(14)
println("a[0] " + a[0])
println("a[1] " + a[1])
println("a[2] " + a[2])
```



Catch an error: expression '**size**' of type 'Int' cannot be invoked as a function. The function 'invoke()' is not found

```
if (args.size() > 0) {  
    println("Args: " + args[0])  
} else {  
    println("Args are empty")  
}
```



Catch an error: expression '**size**' of type 'Int' cannot be invoked as a function. The function 'invoke()' is not found

```
if (args.size() > 0) {  
    println("Args: " + args[0])  
} else {  
    println("Args are empty")  
}
```

```
if (args.isEmpty()) {  
    println("Args: " + args[0])  
} else {  
    println("Args are empty")  
}
```



Catch an error: expecting an element
`println("Args: " + $args[0])`

```
if (args.isNotEmpty()) {  
    println("Args: " + $args[0])  
    println("Args: ${args[0]}")  
    println("Args: ${args.get(0)}")  
} else {  
    println("Args are empty")  
}
```



Catch an error: expecting an element `println("Args: " + $args[0])`

```
if (args.isNotEmpty()) {  
    println("Args: " + $args[0])  
    println("Args: ${args[0]}")  
    println("Args: ${args.get(0)}")  
} else {  
    println("Args are empty")  
}
```

```
if (args.isNotEmpty()) {  
    println("Args: " + args[0])  
    println("Args: ${args[0]}")  
    println("Args: ${args.get(0)}")  
} else {  
    println("Args are empty")  
}
```



Switch-case statements...



Catch an error: expecting '->' **else** "Hi!"

```
if (args.isNotEmpty()) {  
    println("Args: " +  
        "${  
            when (args[0]) {  
                "HELLO_WORLD" -> "Hello"  
                else "Hi!"  
            }  
        }")  
} else {  
    println("Args are empty")  
}
```




Catch an error: expecting '->' else "Hi!"

```
if (args.isNotEmpty()) {  
    println("Args: " +  
        "${  
            when (args[0]) {  
                "HELLO_WORLD" -> "Hello"  
                else "Hi!"  
            }  
        }")  
} else {  
    println("Args are empty")  
}
```

```
if (args.isNotEmpty()) {  
    println("Args: " +  
        "${  
            when (args[0]) {  
                "HELLO_WORLD" -> "Hello"  
                else -> "Hi!"  
            }  
        }")  
} else {  
    println("Args are empty")  
}
```




Interfaces...



Catch an error: 'getHello' hides member of supertype 'Basic' and needs 'override' modifier

```
interface Basic {  
    fun getHello(): String;  
}  
  
class Home : Basic {  
    fun getHello() = "Home"  
}  
  
var a: Home? = Home()  
println("${a?.getHello()}")
```



Catch an error: 'getHello' hides member of supertype 'Basic' and needs 'override' modifier

```
interface Basic {  
    fun getHello(): String  
}  
  
class Home : Basic {  
    fun getHello() = "Home"  
}  
  
var a: Home? = Home()  
println("${a?.getHello()}")
```

```
interface Basic {  
    fun getHello(): String  
}  
  
class Home : Basic {  
    override fun getHello() = "Home"  
}  
  
var a: Home? = Home()  
println("${a?.getHello()}")
```



Catch an error: java.lang.ClassCastException: Fixme\$Home cannot **be cast to Fixme\$Do**

```
interface Relax { fun getAction(): String }
interface Do : Relax { fun getOptions(): String }
class Home : Relax {
    override fun getAction() = "Home"
}
fun getOptions(c: Relax) = if ((c as Do) is Do) {
    c.getOptions()
} else {
    c.getAction()
}
val a = Home()
println(getOptions(a))
```




Catch an error: java.lang.ClassCastException: Fixme\$Home cannot **be cast to Fixme\$Do**

```
interface Relax { fun getAction(): String }
interface Do : Relax { fun getOptions(): String }
class Home : Relax {
    override fun getAction() = "Home"
}
fun getOptions(c: Relax) = if ((c as Do) is Do) {
    c.getOptions()
} else {
    c.getAction()
}
val a = Home()
println(getOptions(a))
```

```
interface Relax { fun getAction(): String }
interface Do : Relax { fun getOptions(): String }
class Home : Relax {
    override fun getAction() = "Home"
}
fun getOptions(c: Relax) = if (c is Do) {
    c.getOptions()
} else {
    c.getAction()
}
val a = Home()
println(getOptions(a))
```




Enums...



Catch an error: expecting ';' after the last **enum** entry or '}'

```
interface TimeOfOperation {  
    fun getTime(): Int  
}  
  
enum class Home(var min: Int) :  
    TimeOfOperation {  
    RELAX(min = 59), CLEAN(min = 58)  
    override fun getTime() = min  
}  
  
fun showOptions(c: TimeOfOperation) {  
    println(c.getTime())  
}  
  
showOptions(Home.CLEAN)
```



Catch an error: expecting ';' after the last **enum** entry or '}'

```
interface TimeOfOperation {  
    fun getTime(): Int  
}  
  
enum class Home(var min: Int) :  
    TimeOfOperation {  
    RELAX(min = 59), CLEAN(min = 58)  
    override fun getTime() = min  
}  
  
fun showOptions(c: TimeOfOperation) {  
    println(c.getTime())  
}  
  
showOptions(Home.CLEAN)
```

```
interface TimeOfOperation {  
    fun getTime(): Int  
}  
  
enum class Home(var min: Int) :  
    TimeOfOperation {  
    RELAX(min = 59), CLEAN(min = 58);  
    override fun getTime() = min  
}  
  
fun showOptions(c: TimeOfOperation) {  
    println(c.getTime())  
}  
  
showOptions(Home.CLEAN)
```



Catch an error: classifier '**Home**' does not have a companion object, and thus must be **initialized** here

```
interface TimeOfOperation {  
    fun getTime(): Int  
}  
  
enum class Home(var min: Int) : TimeOfOperation {  
    RELAX(59), CLEAN(58);  
    override fun getTime() = min  
}
```

```
fun showOptions(c: TimeOfOperation) {  
    println("${c.getTime()} min ")  
}
```

```
showOptions(Home)
```



Catch an error: classifier '**Home**' does not have a companion object, and thus must be **initialized** here

```
interface TimeOfOperation {  
    fun getTime(): Int  
}  
enum class Home(var min: Int) : TimeOfOperation {  
    RELAX(59), CLEAN(58);  
    override fun getTime() = min  
}
```

```
fun showOptions(c: TimeOfOperation) {  
    println("${c.getTime()} min ")  
}
```

```
showOptions(Home)
```

```
interface TimeOfOperation {  
    fun getTime(): Int  
}  
enum class Home(var min: Int) : TimeOfOperation {  
    RELAX(59), CLEAN(58);  
    override fun getTime() = min  
}
```

```
fun showOptions(c: TimeOfOperation) {  
    println("${c.getTime()} min ")  
}
```

```
showOptions(Home.CLEAN)
```

The end

- [linkedin.com/in/evgeny-trushin](https://www.linkedin.com/in/evgeny-trushin)
- github.com/evgeny-trushin/Kotlin_trial-and-error_learning

