Jakieś typy mnie napadły

o systemie typów w Scali 3

i co mi tam do głowy jeszcze przyszło

Przed użyciem kodu ze slajdów zapoznaj się z treścią dokumentacji dołączonej do języka bądź skonsultuj się z doświadczonym programistą, gdyż każdy kod niewłaściwie stosowany zagraża projektowi. To jest demo. W rzeczywistości może istnieć inny, bardziej skuteczny lub efektywny sposób realizacji zadanych funkcjonalności. Przykłady mogą nie stanowić poprawnego kodu w kontekście kompilacji.

Ale ekipa! Bartuś w świetle wujków z Ump

😆

Union & Intersection Types

Intersection Types

nihil novi sub sole

```scala
//Scala 2
def doSth(obj: A with B)

//Scala 3
def doSth(obj: A & B)
```

Union Types

```
val a: A | B = A()
val obj /*: Object*/ = if (???) A() else B()
val aOrB: A | B = if (???) A() else B()
```

Union Types

transparent trait

```
transparent trait T

class T1 extends T
class T2 extends T
```

```
scala> if true then new T1 else new T2
val res0: T1 | T2 = T1@4d525897
```

Union Types

real example

```
class YanushInput:
  def validate: Boolean = ???


class ScexSandbox:
  def validate: Boolean = ???
```

```
Seq(new YanushInput, new ScexSandbox).forall(_.validate)
//compilation error: value validate is not a member of...
```

Value validate is not a member of...

Union Types

proxy pattern?

```
trait HasValidation {
  def validate: Boolean
}

implicit class DropdownValidation(dropdown: YanushInput) extends HasValidation {
  def validate: Boolean = dropdown.validate
}

implicit class ScexValidation(scex: ScexSandbox) extends HasValidation {
  def validate: Boolean = scex.validate
}
```

```
Seq[HasValidation](new YanushInput, new ScexSandbox).forall(_.validate)

Seq[HasValidation](DropdownValidation(new YanushInput), ScexValidation(new
ScexSandbox)).forall(_.validate)
```

Union Types

new approach

```
Seq[YanushInput | ScexSandbox](new YanushInput, new ScexSandbox).forall {
    case x: YanushInput => x.validate
    case x: ScexSandbox => x.validate
}
```

```
Seq(new YanushInput, new ScexSandbox).forall {
    case x: YanushInput => x.validate
    case x: ScexSandbox => x.validate
  }
```

Union Types

extension methods

```scala
extension (x: YanushInput | ScexSandbox)
  def validate: Boolean = x match
    case x: YanushInput => x.validate
    case x: ScexSandbox => x.validate


Seq[YanushInput | ScexSandbox](new YanushInput, new ScexSandbox).forall(_.validate)
Seq(new YanushInput, new ScexSandbox).forall(_.validate)
```

Type

A set of values that can fit into that type

The Type-level Language

- Match Types
- Recursive Types
- Singleton Types
- Arbitrary-sized Tuples
- Type Lambdas
- ...

Tuples

```scala
sealed trait Tuple

case object EmptyTuple extends Tuple

sealed trait NonEmptyTuple extends Tuple

sealed abstract class *:[+H, +T <: Tuple] extends NonEmptyTuple
```

Tuples

Match Types

```scala
type Head[X <: NonEmptyTuple] = X match {
  case x *: _ => x
}
```

Match Types

```scala
type Elem[X] = X match
  case String => Char
  case Array[t] => t
  case Iterable[t] => t
```

Tuples

Recursive Types

```scala
type Last[X <: Tuple] = X match {
  case x *: EmptyTuple => x
  case _ *: xs => Last[xs]
}
```

Tuples

Recursive Types

```scala
type Fold[Tup <: Tuple, Z, F[_, _]] = Tup match
  case EmptyTuple => Z
  case h *: t => F[h, Fold[t, Z, F]]
```

Tuples

Type Lambdas

```scala
type Fold[Tup <: Tuple, Z, F[_, _]] = Tup match
  case EmptyTuple => Z
  case h *: t => F[h, Fold[t, Z, F]]
```

```scala
type Union[T <: Tuple] = Fold[T, Nothing, [x, y] =>> x | y]
```

Singleton Types

intuition

```
def nums = 1 :: 2 :: 3 :: 4 :: Nil
def map[A, B](ls: List[A], f: A => B): List[B] =
  ls match
    case Nil => Nil
    case h :: t => f(h) :: map(t, f)
def mapped = nums.map(i => i + 10)
def filtered = mapped.filter(i => i < 14)
def result = fold(filtered, "", (i, acc) => toString(i) ++ acc)
```

Singleton Types

intuition

```
type Nums = 1 *: 2 *: 3 *: 4 *: EmptyTuple
def map[A, B](ls: List[A], f: A => B): List[B] =
  ls match
    case Nil => Nil
    case h :: t => f(h) :: map(t, f)
def mapped = nums.map(i => i + 10)
def filtered = mapped.filter(i => i < 14)
def result = fold(filtered, "", (i, acc) => toString(i) ++ acc)
```

Singleton Types

intuition

```
type Nums = 1 *: 2 *: 3 *: 4 *: EmptyTuple
type Map[T <: Tuple, F[_]] <: Tuple =
  T match
    case EmptyTuple => EmptyTuple
    case h *: t => F[h] *: Map[t, F]
def mapped = nums.map(i => i + 10)
def filtered = mapped.filter(i => i < 14)
def result = fold(filtered, "", (i, acc) => toString(i) ++ acc)
```

Singleton Types

intuition

```
type Nums = 1 *: 2 *: 3 *: 4 *: EmptyTuple
type Map[T <: Tuple, F[_]] <: Tuple =
  T match
    case EmptyTuple => EmptyTuple
    case h *: t => F[h] *: Map[t, F]
type Mapped = Nums `Map` ([X] =>> X + 10)
def filtered = mapped.filter(i => i < 14)
def result = fold(filtered, "", (i, acc) => toString(i) ++ acc)
```

Singleton Types

intuition

```
type Nums = 1 *: 2 *: 3 *: 4 *: EmptyTuple
type Map[T <: Tuple, F[_]] <: Tuple =
  T match
    case EmptyTuple => EmptyTuple
    case h *: t => F[h] *: Map[t, F]
type Mapped = Nums `Map` ([X] =>> X + 10)
type Filtered = Mapped `Filter` ([X] =>> X < 14)
def result = fold(filtered, "", (i, acc) => toString(i) ++ acc)
```

Singleton Types

intuition

```
type Nums = 1 *: 2 *: 3 *: 4 *: EmptyTuple
type Map[T <: Tuple, F[_]] <: Tuple =
  T match
    case EmptyTuple => EmptyTuple
    case h *: t => F[h] *: Map[t, F]
type Mapped = Nums `Map` ([X] =>> X + 10)
type Filtered = Mapped `Filter` ([X] =>> X < 14)
type Result = Fold[Filtered, "", [I, Acc] =>> ToString[I] + Acc]
```

Singleton Types

intuition

```
type Nums = 1 *: 2 *: 3 *: 4 *: EmptyTuple
type Map[T <: Tuple, F[_]] <: Tuple =
  T match
    case EmptyTuple => EmptyTuple
    case h *: t => F[h] *: Map[t, F]
type Mapped = Nums `Map` ([X] =>> X + 10)
type Filtered = Mapped `Filter` ([X] =>> X < 14)
type Result = Fold[Filtered, "", [I, Acc] =>> ToString[I] + Acc]
```

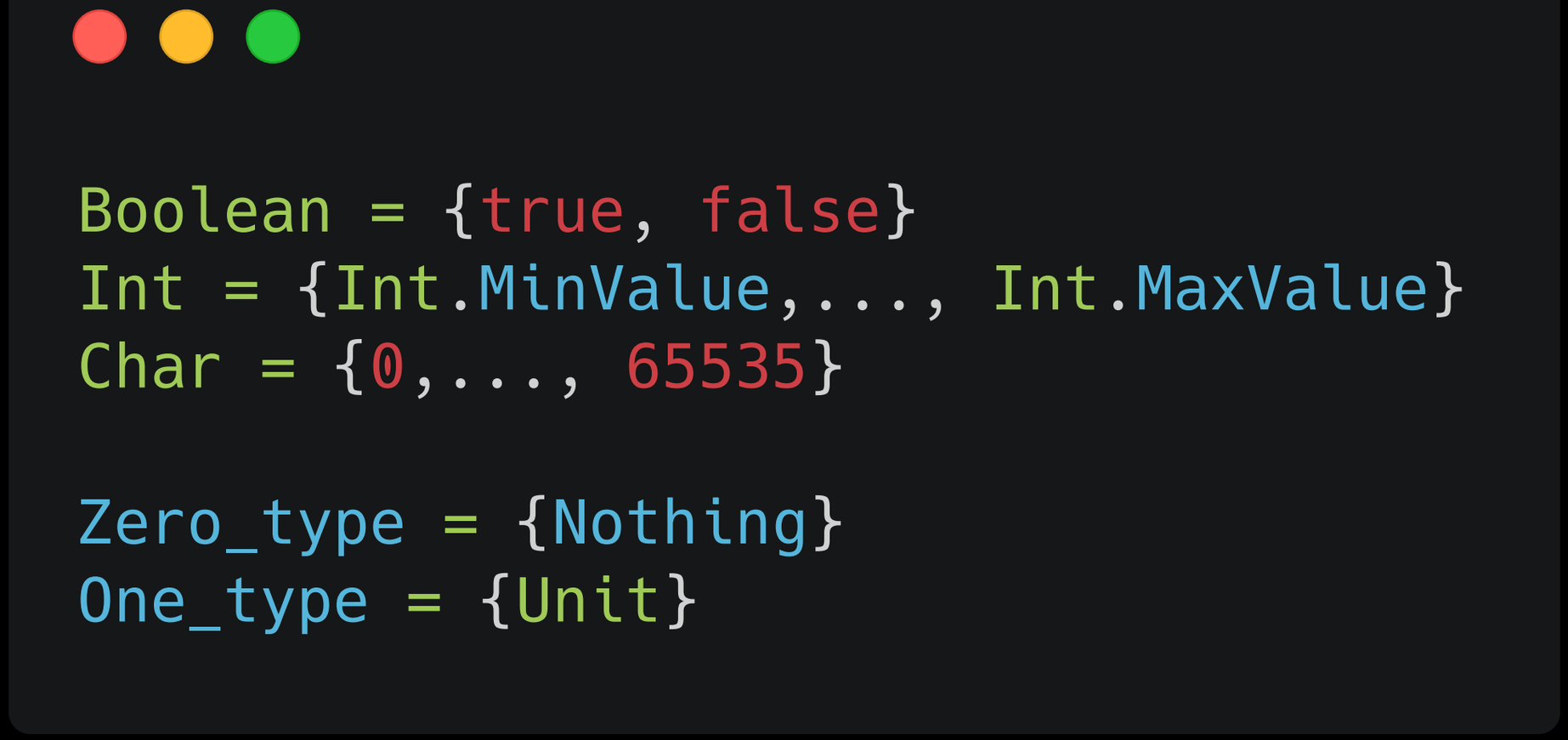
```
scala> constValue[Result]
val res0: String = 111213
```

Singleton Type

A type that has only one member.

Type

A set of values that can fit into that type


Boolean = {true, false}
Int = {Int.MinValue,..., Int.MaxValue}
Char = {0,..., 65535}
Zero_type = {Nothing}
One_type = {Unit}

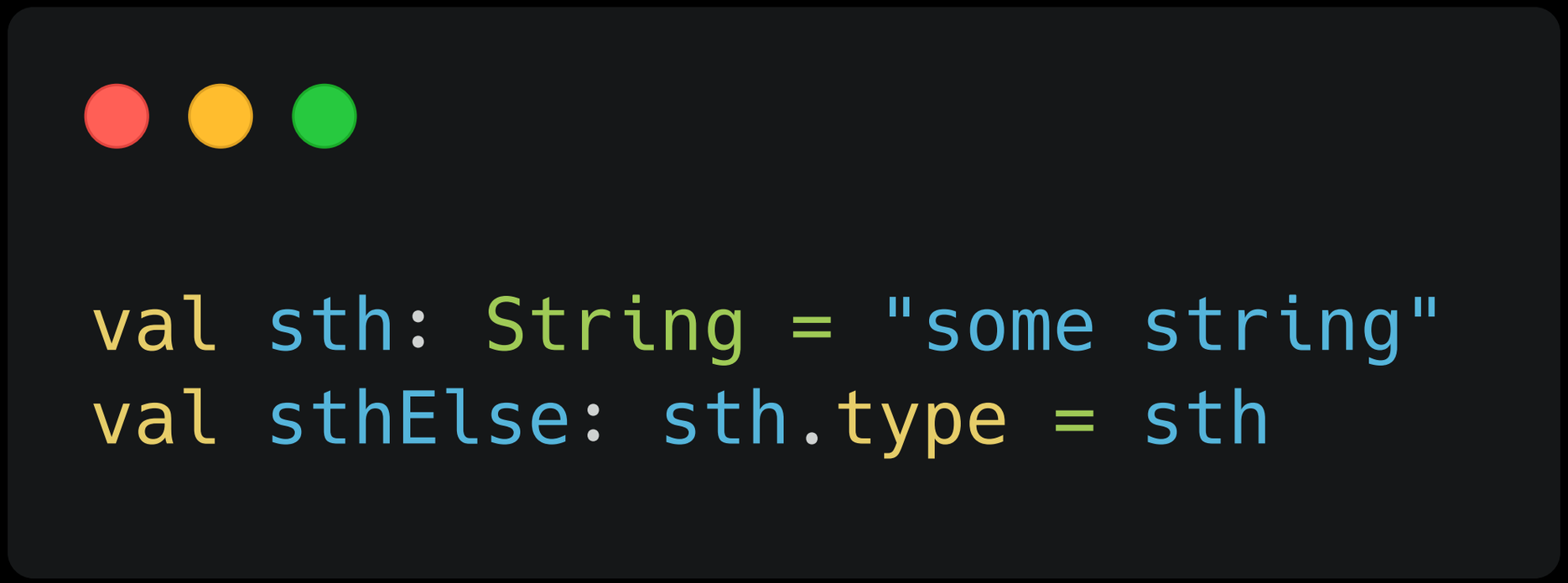
```
val a: Unit = ()
println() == () //always true
```

```
object someObject:
  val ref: someObject.type = someObject
```

```scala
class Parent:
  object Child

def f(another: Parent): Child.type = another.Child
```

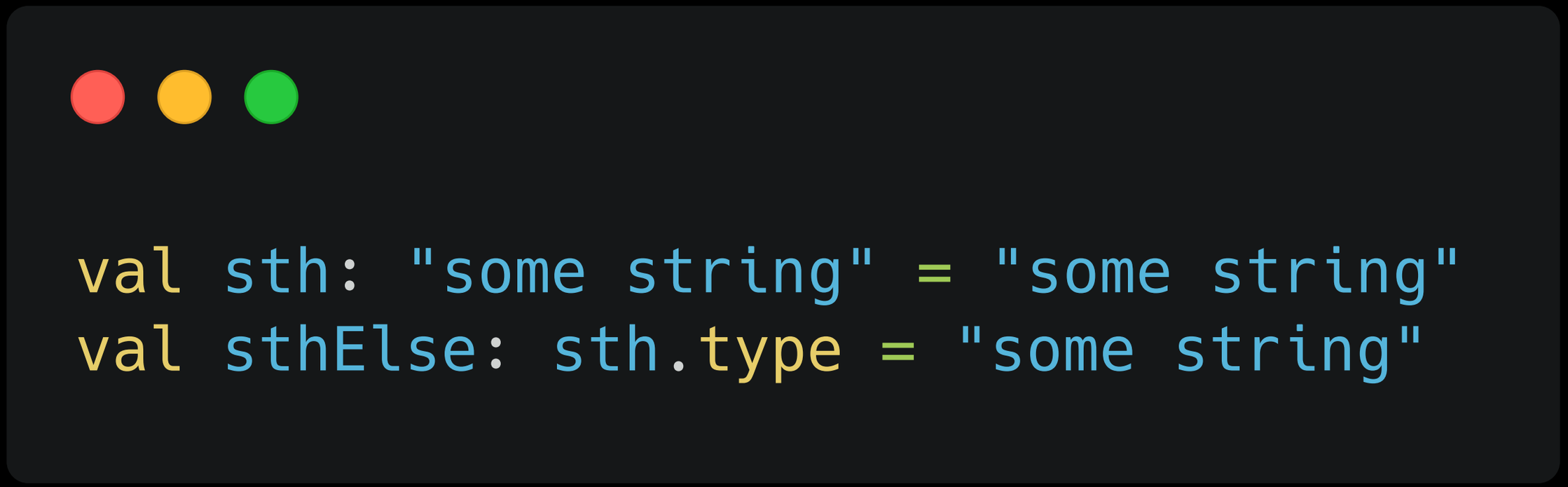
Found: another.Child.type
Required: Parent.this.Child.type

```
class Parent:
  object Child

  def f(another: Parent): another.Child.type = another.Child
```


val sth: String = "some string"
val sthElse: sth.type = sth

```
val sth: String = "some string"
val sthElse: sth.type = "some string"
```

Found: ("some string" : String)
Required: (halotukozak.singleton.value.sth : String)



```
val sth: "some string" = "some string"
val sthElse: sth.type = "some string"
```

```
val sth = "some string"
val sthElse: sth.type = "some string"
```

Found: ("some string" : String)
Required: (halotukozak.singleton.value.sth : String)

```
final val sth = "some string"
val sthElse: sth.type = "some string"
```

```scala
def identity[A](x: A): A = x

def identity(x: Any): x.type = x

val id: Int = identity(42)
```

```scala
def identity[A](x: A): A = x
```

```scala
def identity(x: Any): x.type = x

val id: 42 = identity(42)
```

Non-live coding

```scala
type HeadingLevel = 1 | 2 | 3 | 4 | 5 | 6

inline def heading[N <: HeadingLevel : ValueOf](inline inner: String): String =
    "#" * valueOf[N] + inner
```

```
assert(heading[1]("Hello") == "#Hello")
assert(heading[2]("Hello") == "##Hello")
assert(heading[3]("Hello") == "###Hello")
assert(heading[4]("Hello") == "####Hello")
assert(heading[5]("Hello") == "#####Hello")
assert(heading[6]("Hello") == "######Hello")


assertDoesNotCompile("""heading[7]("Hello")""")
assertDoesNotCompile("""heading[0]("Hello")""")
assertDoesNotCompile("""heading[-5]("Hello")""")
```

```scala
inline def link[Title <: String & Singleton]
  (inline title: Title, inline url: String)
  (using Matches[Title, ".*MUST.*"] =:= true)
: String = s"[$title]($url)"
```

```
assert(link(title = "you MUST see", url = "https://www.avsystem.com") ==
       "[you MUST see](https://www.avsystem.com)")

assertDoesNotCompile("""link(title = "you SHOULD see", url = "https://www.avsystem.com")""")
assertDoesNotCompile("""link(title = "have to you see", url = "https://www.avsystem.com")""")
```

Opaque Type Aliases

```
object Definitions:
  type Alias = String

object Usage:
  val a: Alias = "a"
  val b: String = a
```

```
object Definitions:
  opaque type Alias = String

object Usage:
  val a: Alias = "a"
  val b: String = a
```

Found: ("a" : String)
Required: halotukozak.Definitions.Alias

Found: (a : halotukozak.Definitions.Alias)
Required: String

```scala
object Definition:
  opaque type Complex = (Double, Double)

  object Complex:
    def apply(real: Double, imag: Double): Complex = (real, imag)

  extension (c: Complex)
    def +(that: Complex): Complex = (c._1 + that._1, c._2 + that._2)

object impl:
  val a = Complex(1, 2)
  val b = Complex(3, 4)
  val sum = a + b
```

```
val a = Complex(1, 2)
val real: Double = a._1
```

value _1 is not a member of halotukozak.opaque.Definition.Complex

```scala
opaque type Complex <: (Double, Double) = (Double, Double)

val real: Double = a._1
val imag: Double = a._2

val sum = a + (3, 4)
```

Found: (Int, Int)
Required: halotukozak.opaque.Definition.Complex

Type Lambdas

Type

A set of values that can fit into that type

Level 0

Value Types

```
val a: Int = 42
val b: String = "coffee"

class Film(title: String)

val c: Film = Film("Dune")
```

Level 1

Generics (type constructors)

```scala
val d: List[Int] = List(1, 2, 3)

class Box[A](value: A)

val e: Box[List[Boolean]] = Box(List(true, false, true))
```

Attention,

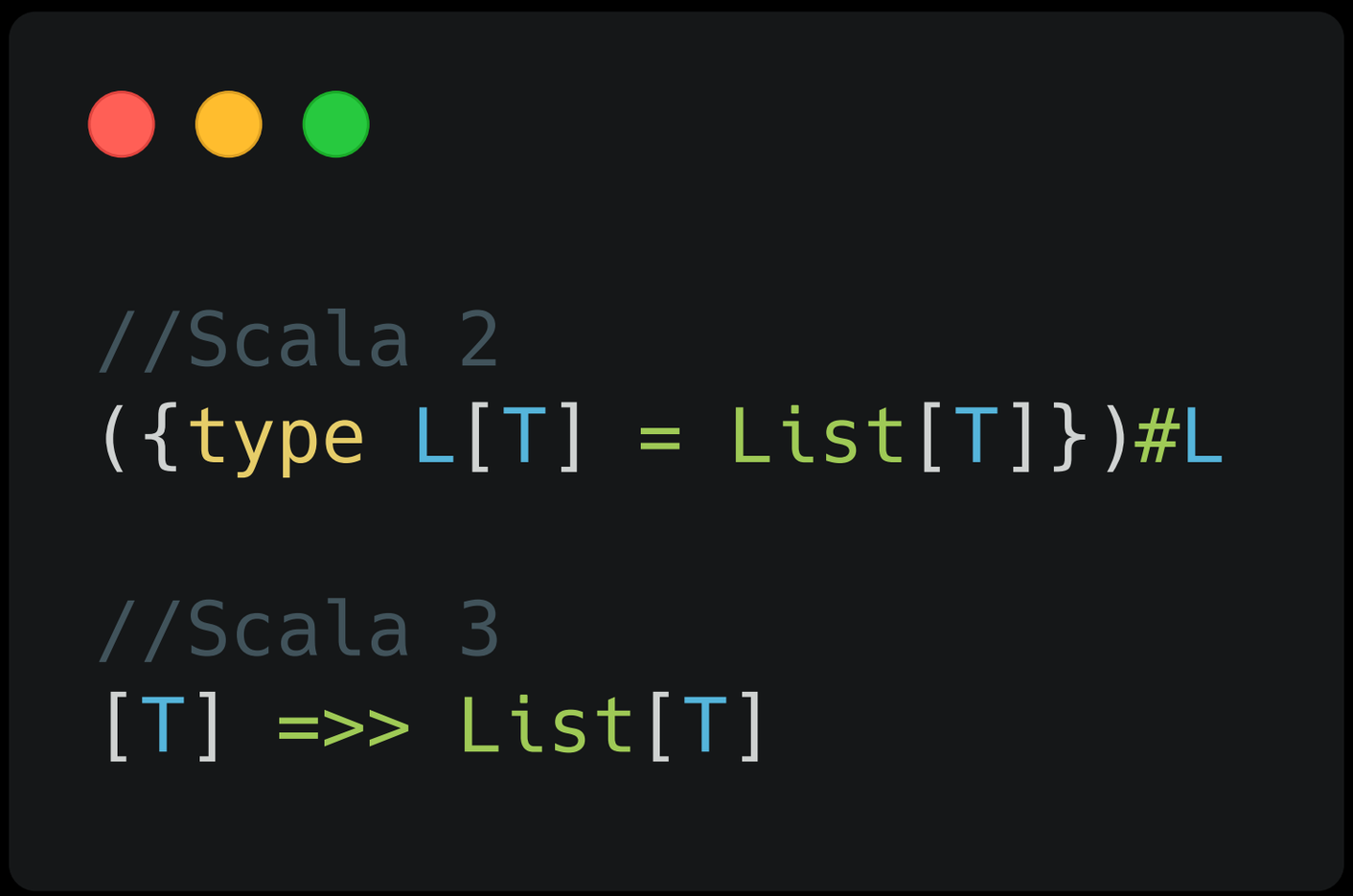
Functors!

Level 2+

Higher-Kinded Types

```scala
class Functor[F[_]]

val fList = new Functor[List]
val fOption = new Functor[Option]

class Hell[F[_[_]]]

val hellFunctor = new Hell[Functor]
```

```scala
trait Functor[A, +M[_]] {
  def map[B](f: A => B): M[B]
}

case class SeqFunctor[A](seq: Seq[A]) extends Functor[A, Seq] {

  def map[B](f: A => B): Seq[B] = seq.map(f)
}

val mappedSeq: Seq[Int] = SeqFunctor(Seq(1, 2, 3)).map(_ + 1)

case class MapFunctor[K, V](mapKV: Map[K, V])
    extends Functor[V, ({type L[a] = Map[K, a]})#L] {

  def map[V2](f: V => V2): Map[K, V2] = mapKV map { case (k, v) => (k, f(v)) }
}

val mappedMap: Map[Int, Int] = MapFunctor(Map(1 -> 1, 2 -> 2, 3 -> 3)).map(_ + 1)
```



```scala
//Scala 2
({type L[T] = List[T]})#L

//Scala 3
[T] =>> List[T]
```


[T] =>> List[T]
[L, R] =>> Either[L, R]
[F[_]] =>> F[Int]

```scala
trait Functor[A, +M[_]]:
  def map[B](f: A => B): M[B]

case class MapFunctor[K, V](mapKV: Map[K, V]) extends Functor[V, [T] =>> Map[K, T]]:
  def map[V2](f: V => V2): Map[K, V2] = mapKV map { case (k, v) => (k, f(v)) }

val mappedMap: Map[Int, Int] = MapFunctor(Map(1 -> 1, 2 -> 2, 3 -> 3)).map(_ + 1)
```

Polymorphic Function Types

a function type which accepts type parameters

Polymorphic Function Types

```scala
object Definition:
  type Fold[A, B] = (B, (A, B) => B) => B

  val Nil: [A, B] => () => Fold[A, B] =
    [A, B] => () => (empty, f) => empty

  val List: [A, B] => (A, Fold[A, B]) => Fold[A, B] =
    [A, B] => (head: A, tail: Fold[A, B]) => (empty, f) => f(head, tail(empty, f))

object Usage:
  val list: [B] => () => Fold[Int, B] =
    [B] => () => List(1, List(2, List(3, Nil())))

  val sum = list()(0, (a, b) => a + b)
  val product = list()(1, _ * _)
```

Macros

a piece of code executed by the compiler. It is is possible to analyze and generate code

Inline

```scala
inline def heading[N <: HeadingLevel : ValueOf]
    (inline inner: String): String =
  "#" * valueOf[N] + inner
```

Inline

```scala
inline def f(p: Boolean) = if p then "true" else 0

val x: String | Int = f(true)

transparent inline def f2(p: Boolean) = if p then "true" else 0

val y: String = f2(true)
val z: Int = f2(false)
```

Inline

```
inline def f3(p: Boolean) = inline if p then "true" else 0
transparent inline def f4(o: String) = inline o match
  case "Kotlin" => "amazing"
  case "Scala" => 10d
  case _ => 0

val b = f3(math.pow(3.0, 2.0) == 9.0)
val c: String = f4("Kotlin")
val d: Double = f4("Scala")
val e: Int = f4("Java")
```

Cannot reduce `inline if` because its condition is not a constant value

Multi-Staging

Quotation & Splicing

```scala
inline def helloWord: String = ${ helloWordImpl }
def helloWordImpl(using Quotes): Expr[String] = '{ "Hello World" }

inline def printUpperCase(str: String): Unit = ${ printUpperCaseImpl('{ str }) }
def printUpperCaseImpl(str: Expr[String])(using Quotes): Expr[Unit] = '{ println($str.toUpperCase) }
```

Multi-Staging

Level consistency

```scala
inline def unrolledPower(x: Double, n: Int): Double = ${ unrolledPowerImpl('{ x }, '{ n }) }
def unrolledPowerImpl(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
  n match
    case '{ 0 } => '{ 1.0 }
    case '{ 1 } => x
    case _ => '{ $x * ${ unrolledPowerImpl(x, '{ $n - 1 }) } }
```

Macros

Non-real example

```
text[Normal]("HelloWorld") shouldBe "HelloWorld"
text[Bold]("HelloWorld") shouldBe "**HelloWorld**"
text[Bold & Italic]("HelloWorld") shouldBe "***HelloWorld***"
text[Italic & Bold]("HelloWorld") shouldBe "***HelloWorld***"
text[Strikethrough]("HelloWorld") shouldBe "~~HelloWorld~~"
text[Code]("HelloWorld") shouldBe "`HelloWorld`"
text[Code] {
  """
    |var a = 1
    |var b = 2
    |""".stripMargin
} shouldBe {
  """```
    |
    |var a = 1
    |var b = 2
    |
    |```""".stripMargin
}
text[Bold]("Hello" + text[Italic]("World")) shouldBe "**Hello*World***"
text[Strikethrough]("Hello" + text[InlineCode]("World")) shouldBe "~~Hello`World`~~"
text[Bold](text[Bold](text[Bold]("Hello"))) shouldBe "******Hello******"
```

Macros

Non-real example

```
type Normal
type Bold
type Italic
type Strikethrough
type InlineCode
type BlockCode
type Code = InlineCode | BlockCode

type TextStyle = Normal | Bold | Italic | Strikethrough | Code

inline def text[Style <: TextStyle](inline inner: String): String = ${ textImpl[Style]('{ inner }) }
private def textImpl[Style <: TextStyle : Type](inner: Expr[String])(using Quotes): Expr[String] = {
  Type.of[Style] match {
    case '[Normal] => inner
    case '[Bold & Italic] => '{ "***" + $inner + "***" }
    case '[Bold] => '{ "**" + $inner + "**" }
    case '[Italic] => '{ "*" + $inner + "*" }
    case '[Strikethrough] => '{ "~~" + $inner + "~~" }
    case '[InlineCode] => '{ "`" + $inner + "`" }
    case '[BlockCode] => '{ "```\n" + $inner + "\n```" }
    case '[Code] => '{
      if ($inner.split("\n").length == 1) "`" + $inner + "`"
      else "```\n" + $inner + "\n```"
    }
  }
}
```

Macros

Safety

Macros

Safety

Hygiene

Well-typed

Type consistency

Macros Safety

Hygiene

```scala
inline def debug(expr: => Any): Unit = ${ debugImpl('{ expr }) }

def debugImpl(expr: Expr[Any])(using qctx: Quotes): Expr[Unit] = {
  val code: String = expr.show
  '{
    println(s"Debugging: $code")
  }
}
```

access to value code from wrong staging level:
- the definition is at level 0,
- but the access is at level 1.

Macros Safety

Well-typed

```
inline def isEven(inline n: Int): Boolean = ${ isEvenImpl('{ n }) }
```

```
def isEvenImpl(n: Expr[String])(using Quotes): Expr[Int] = {
  val result = '{ 2 == 0 }
  result
}
```

Macros Safety

Type consistency

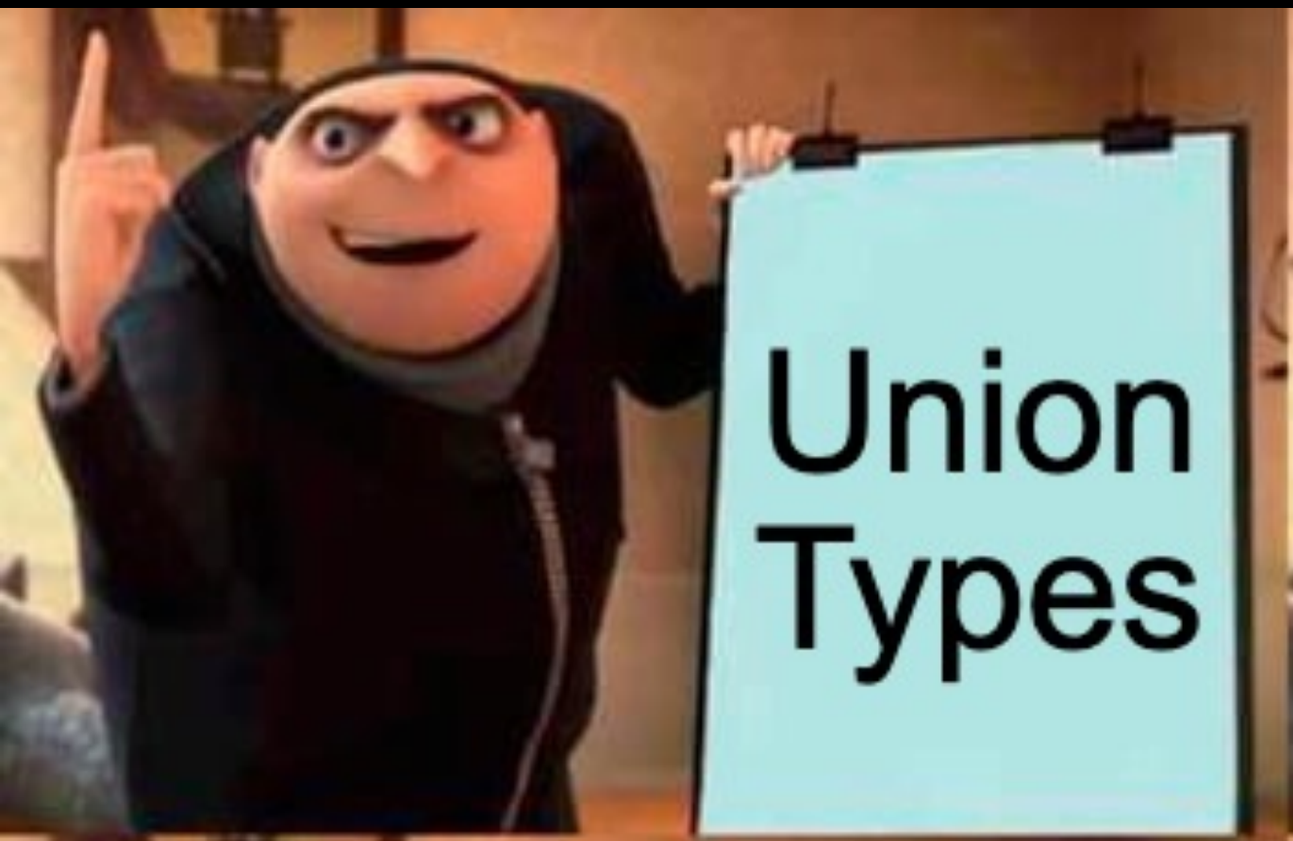
```scala
def evalAndUse[T](x: Expr[T])(using Quotes) = '{
  val x2: T = $x
  ???
}
```

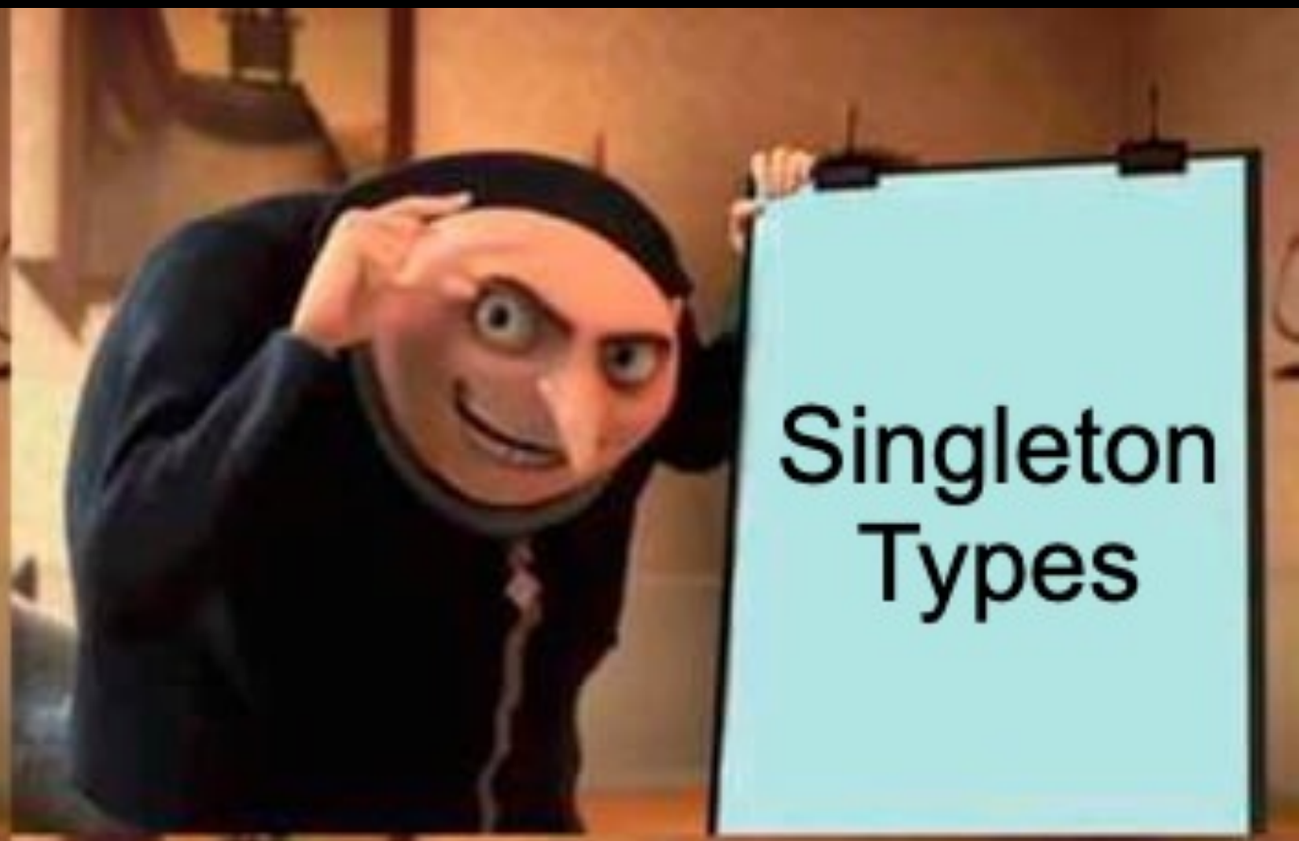
Reference to T within quotes requires a given scala.quoted.Type[T] in scope.

Macros Safety

Type consistency

```
def evalAndUse[T: Type](x: Expr[T])(using Quotes) = '{
  val x2: T = $x
  ???
}
```


Union
Types


Singleton
Types


Macros
imgflip.com


Macros

A really long slide with links which will never be visited

- Scala 3 Reference
- Custom Compile-Time Errors with a Vengeance (Daniel Beskin)
- Scala 3 Macros without Pain (Pawel L.)
- Type Erasure in Scala (Sid Shanker)
- TASTY way of (re)writing macros in Scala 3 (Kacper Korban)
- intro to Scala 3 macros (eugene yokota)
- Any call to technical support (HRejterzy)
- My useless library for markdown generation with Scala
- The code from these slides
- My audience when the topic of functors came up on the screen
- Scala 3: Opaque Types (Daniel Ciocîrlan)
- Singleton types (Jakub Kozłowski)
- Implementing a Macro (Nicolas Stucki)
- Metaprogramming in Scala 2 & 3 (Jan Chyb)
- Functional Programming Strategies In Scala with Cats (Noel Welsh)
- Polymorphic Function Types in Scala 3 (Guillaume Martres)