Type-Level Programming 101

The Subspace of Scala

Joe Barnes

Follow along at http://172.30.7.195:8080

Powered by

Joe Barnes

@joescii

prose :: and :: conz

Primarily Java from 2004-2013

Started Scala late 2012

Started Scala at Mentor Graphics late 2013

Lift committer in July 2014

So... Type programming...

What do *you* know about *that*, Joe??

Not much…

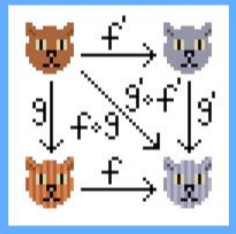
So… Type programming…

What do *you* know about *tha*

I've made zero contributions to shapeless

Not much...

Also no contributions to Cats



Type-Level Programming 101

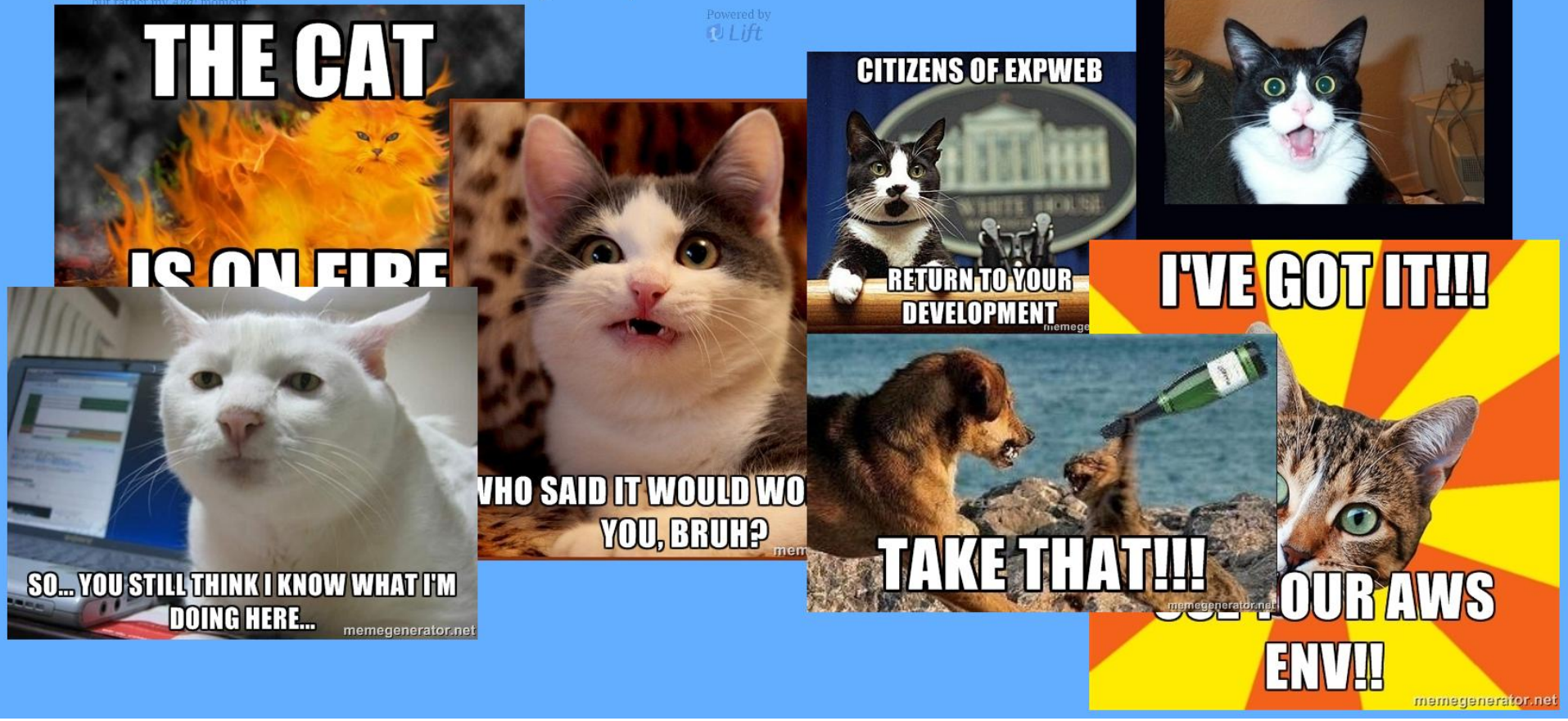
The Subspace of Scala

Joe Barnes

Follow along at http://172.30.7.195:8080

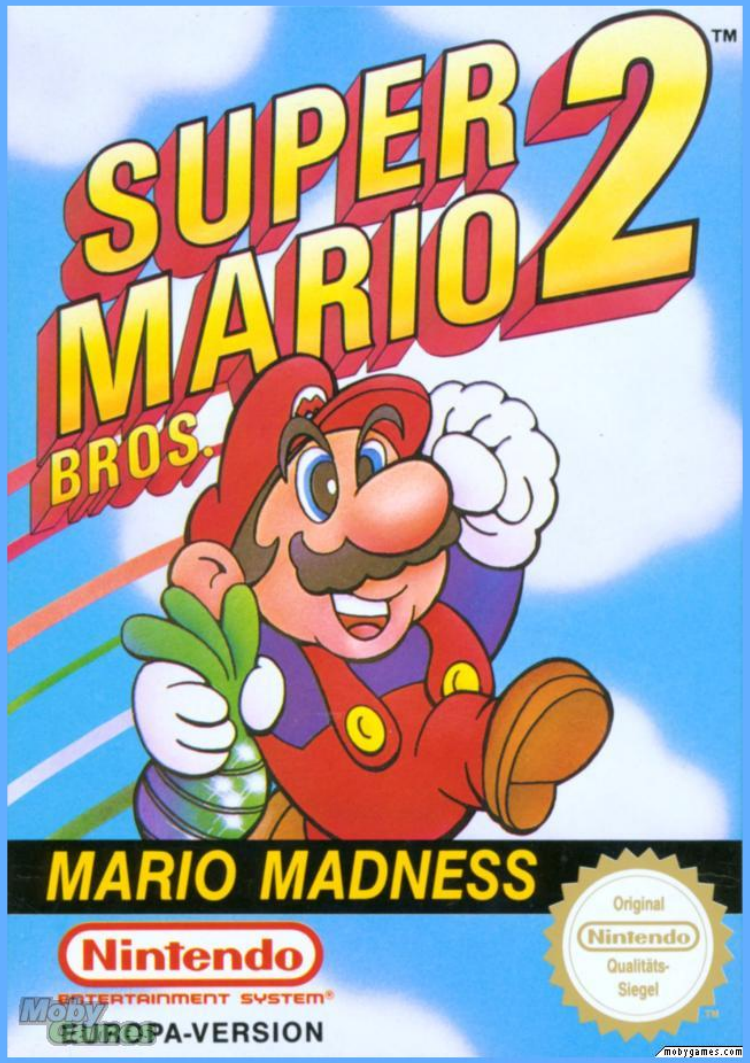
Powered by Lift

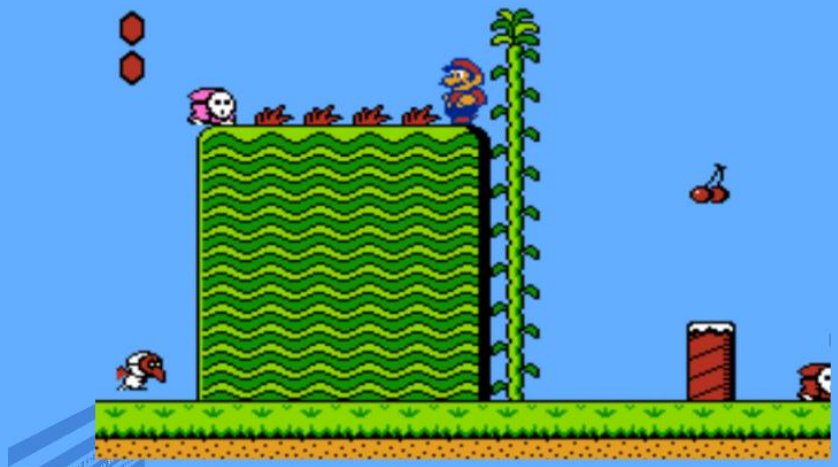
Although I have contributed plenty of cat memes!



I'm not here to share expertise on the subject,
but rather my *Aha!* moment.

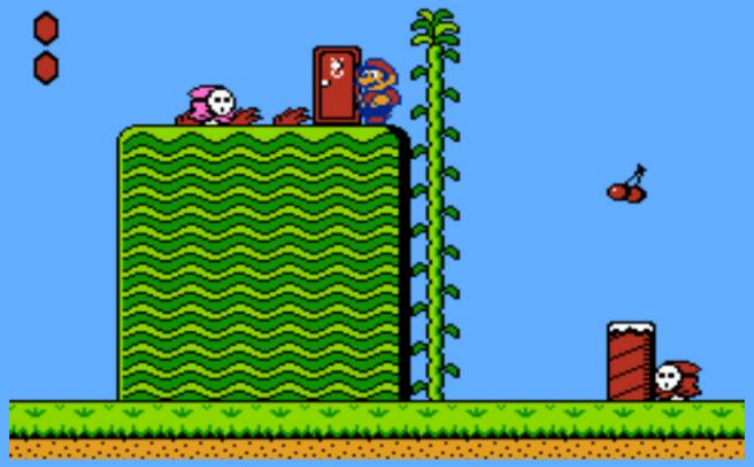
Programming in Scala is like...


SUPER MARIO 2
BROS.
MARIO MADNESS
Nintendo
ENTERTAINMENT SYSTEM
EUROPA-VERSION
Original
Nintendo
Qualitäts-
Siegel
mobygames.com



Normal value programming

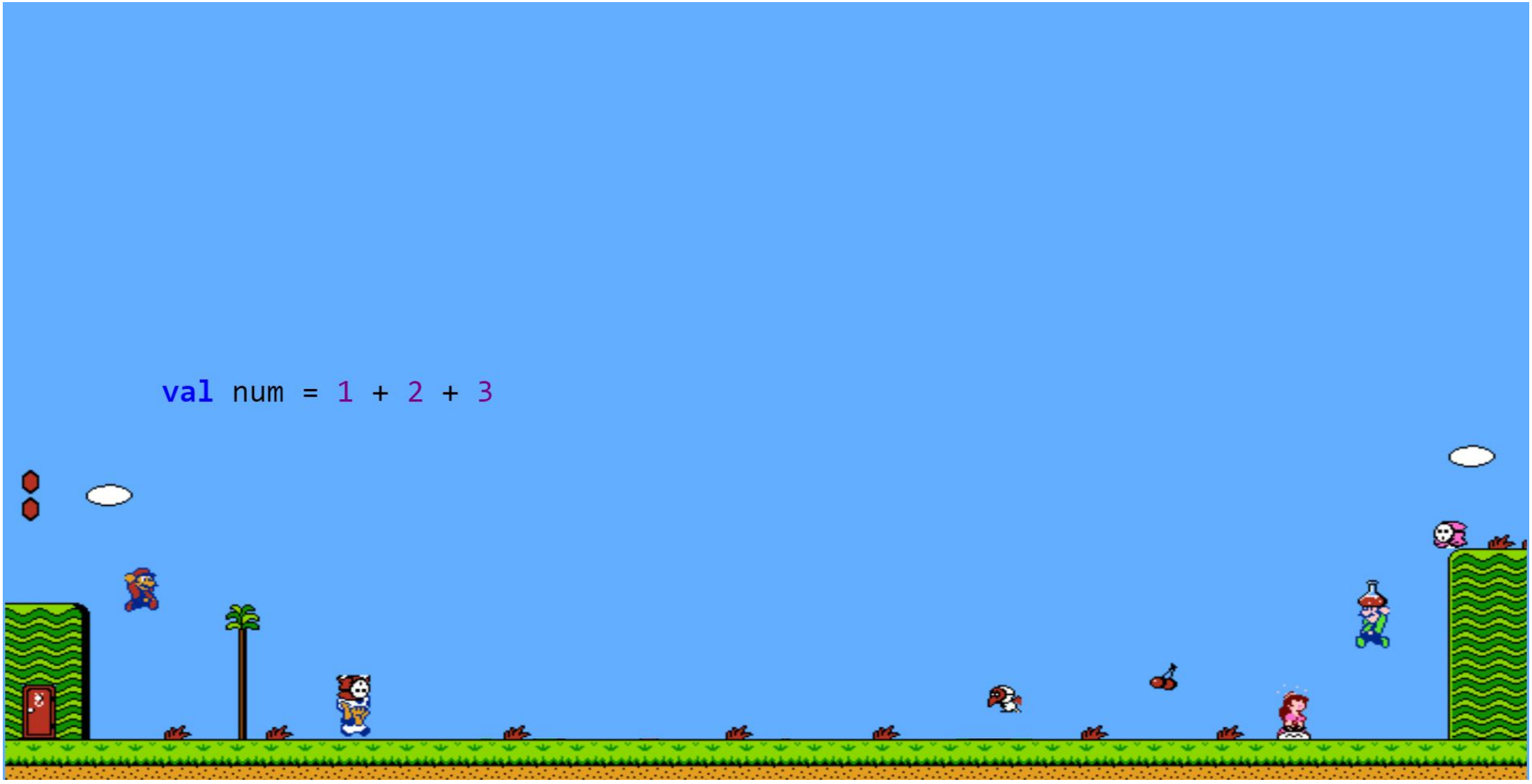
A flask of type programming!



Where does this go?

SUBSPACE!

```
val num = 1 + 2 + 3
```



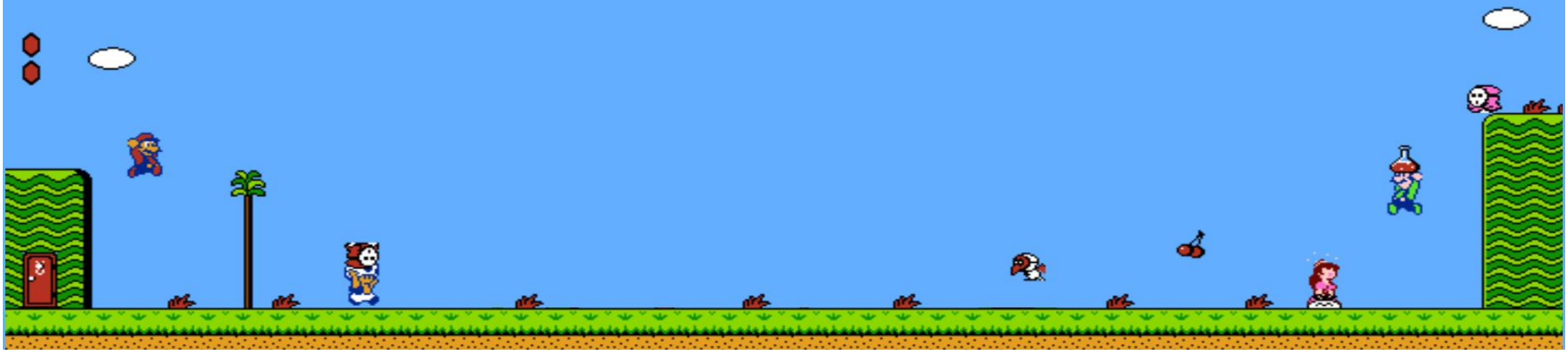
```
val num = 1 + 2 + 3
```

```
lazy val str = "a"+"b"+"c"
```

```
val num = 1 + 2 + 3
```

```
lazy val str = "a"+"b"+"c"
```

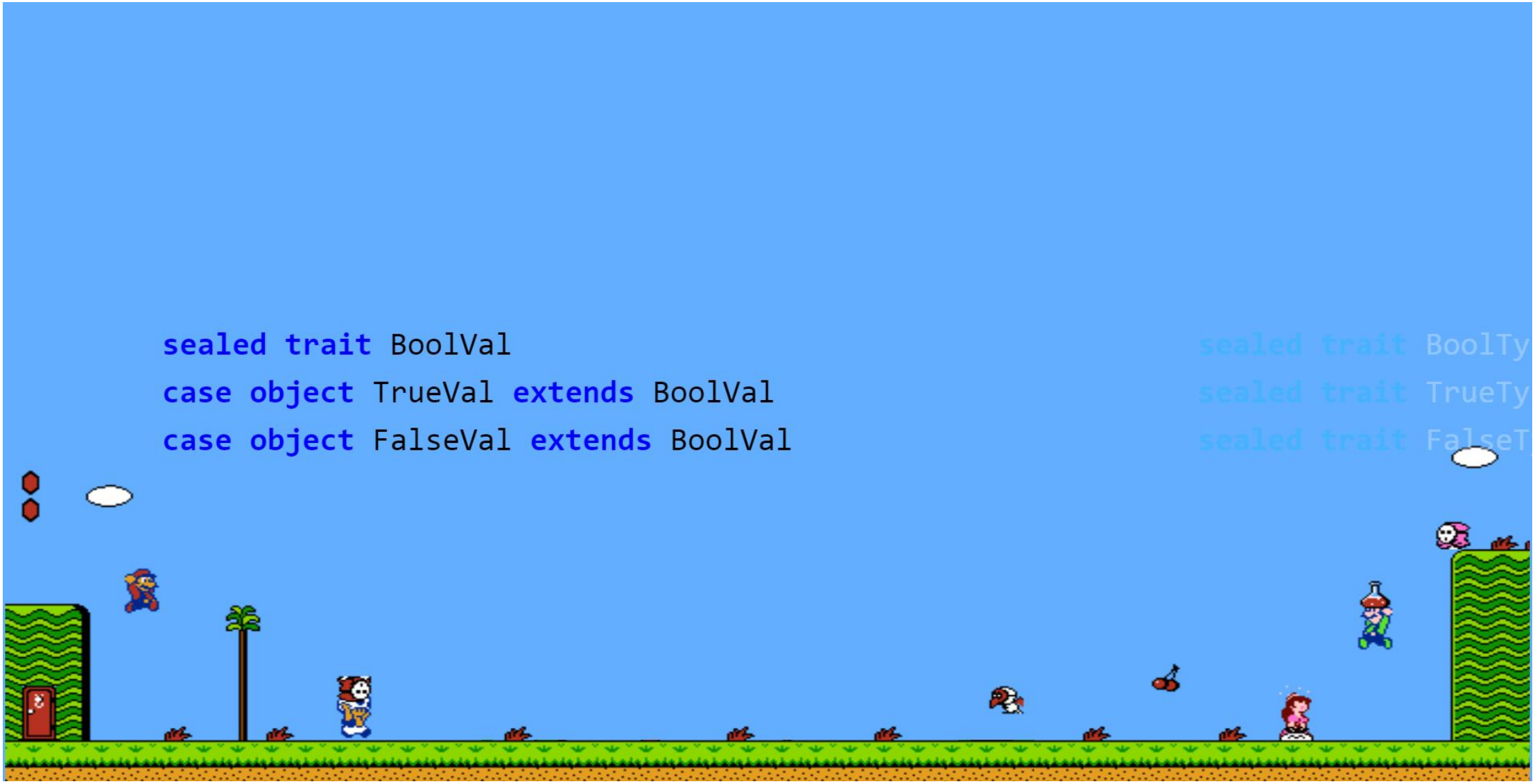
```
def now = new java.util.Date
```



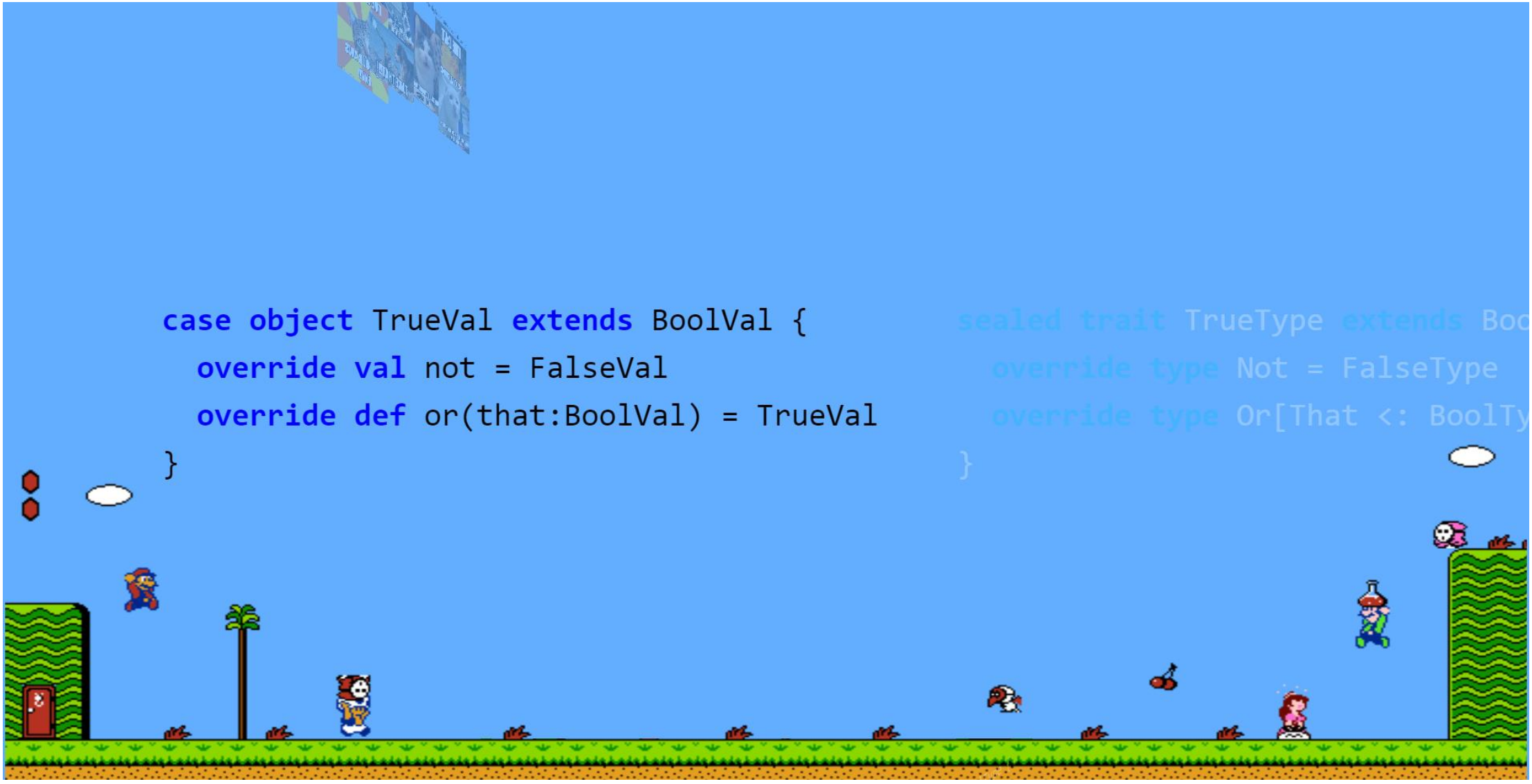
```
type MyMap = Map[Int, String]
```

Let's compare values to types with the simplest domain ever

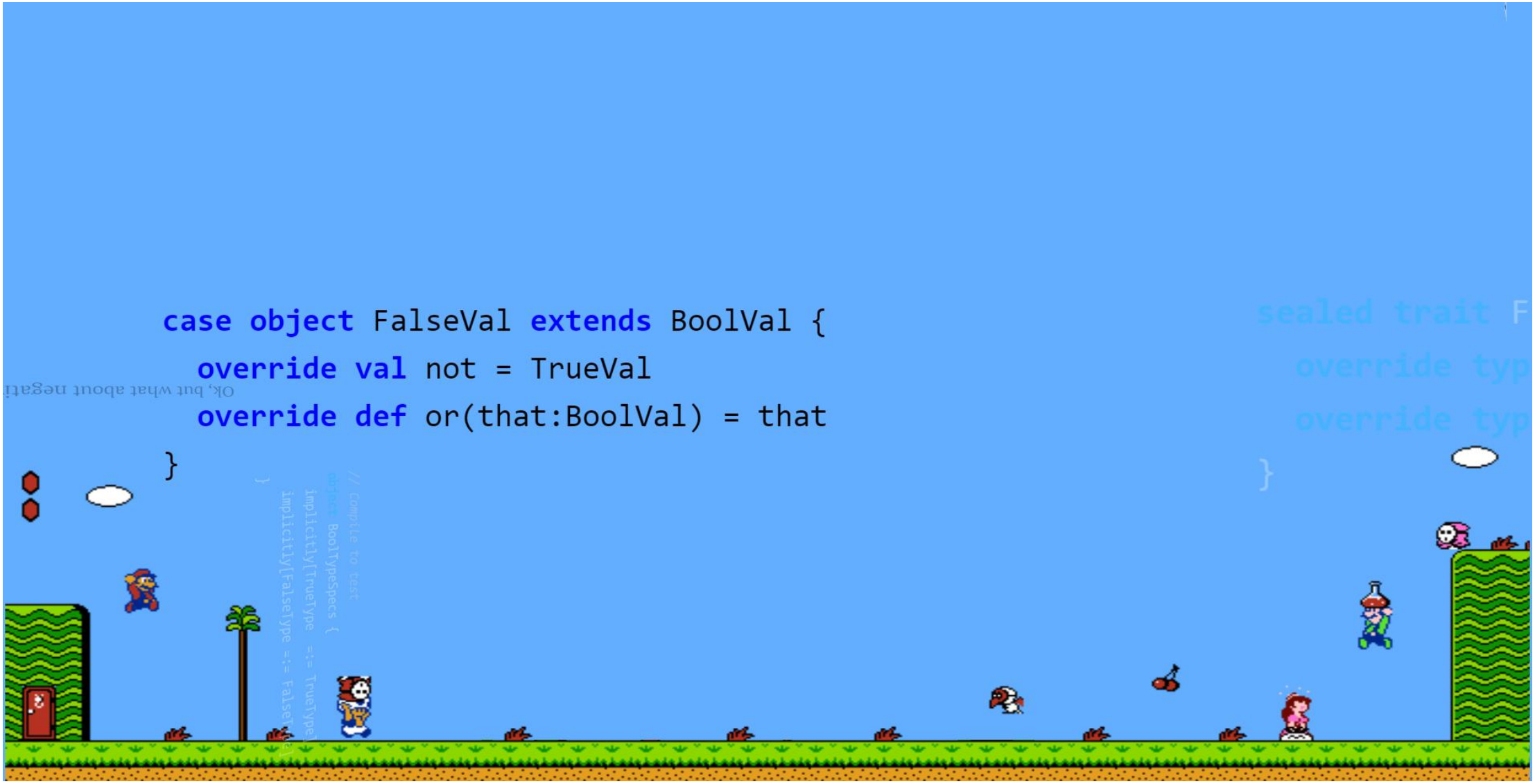
```scala
sealed trait BoolVal
case object TrueVal extends BoolVal
case object FalseVal extends BoolVal
```



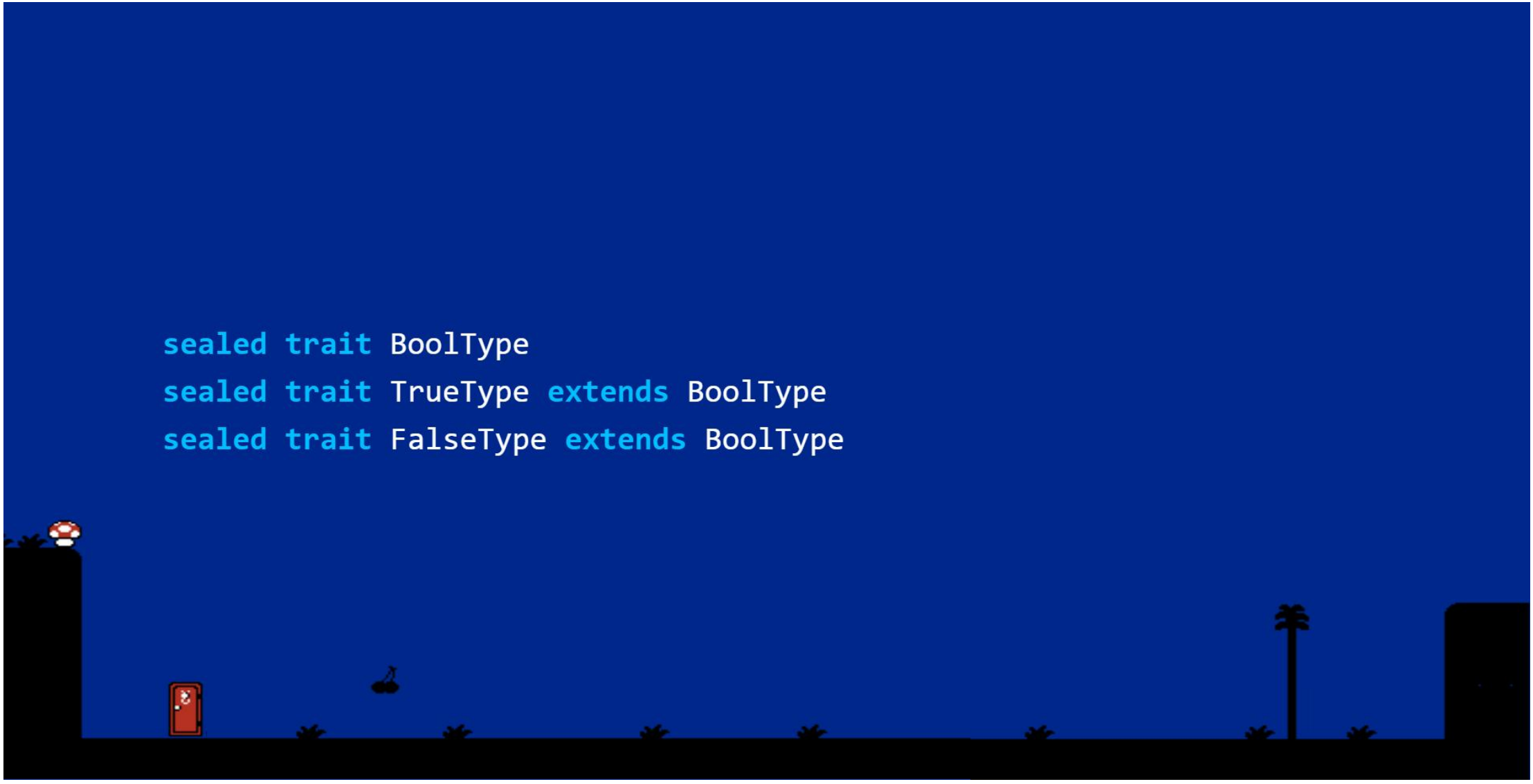

sealed trait BoolVal {
def not:BoolVal
def or(that:BoolVal):BoolVal
}
sealed trait BoolType {
type Not <: BoolType
type Or[That <: BoolType] <:
}


case object TrueVal extends BoolVal {
override val not = FalseVal
override def or(that:BoolVal) = TrueVal
}
sealed trait TrueType extends Boo
override type Not = FalseType
override type Or[That <: BoolTy
}

```
case object FalseVal extends BoolVal {
  override val not = TrueVal
  override def or(that:BoolVal) = that
}
```



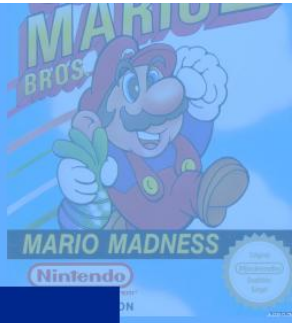
```scala
sealed trait BoolType
sealed trait TrueType extends BoolType
sealed trait FalseType extends BoolType
```



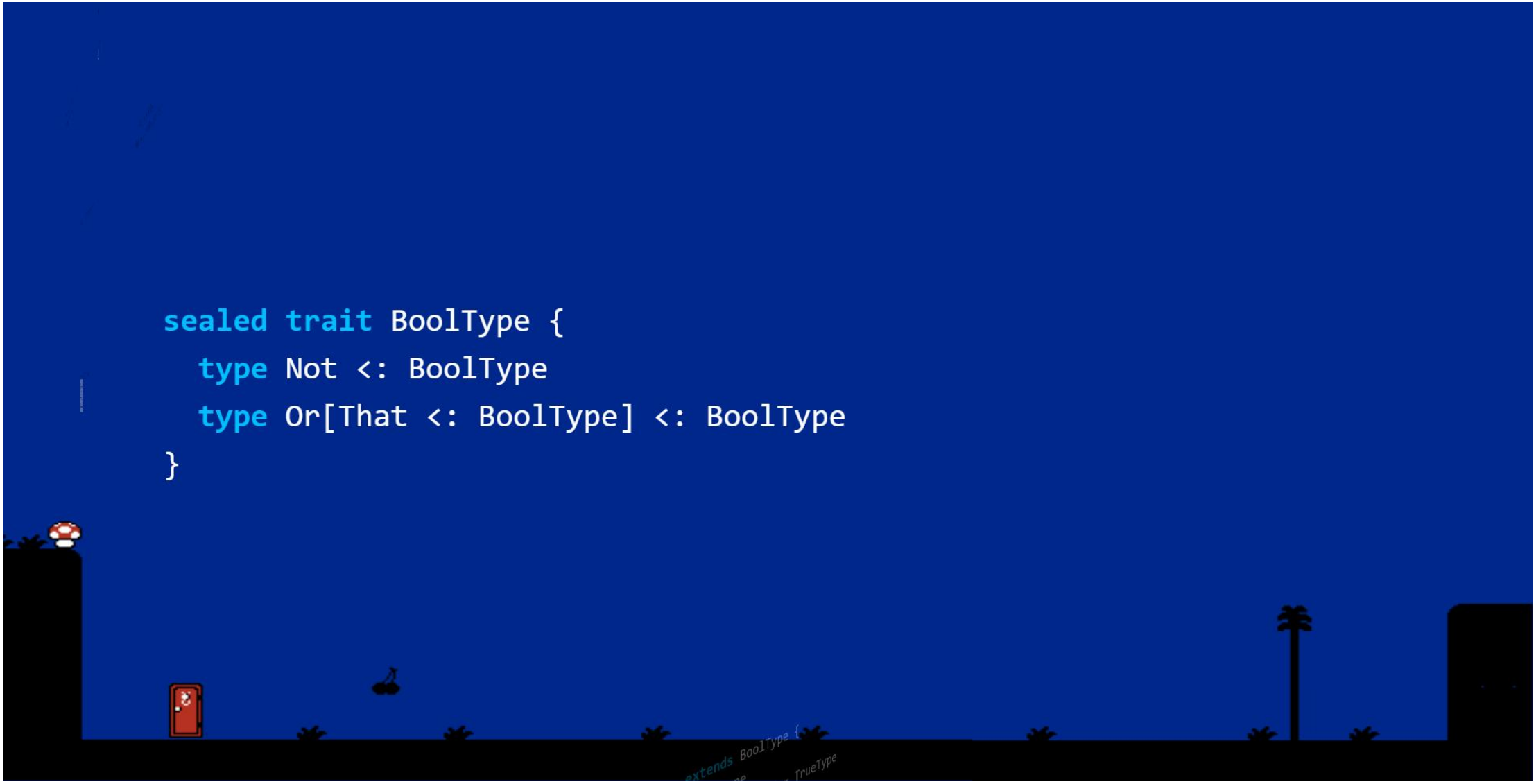
oe Barnes
@joescii
rose :: and :: conz

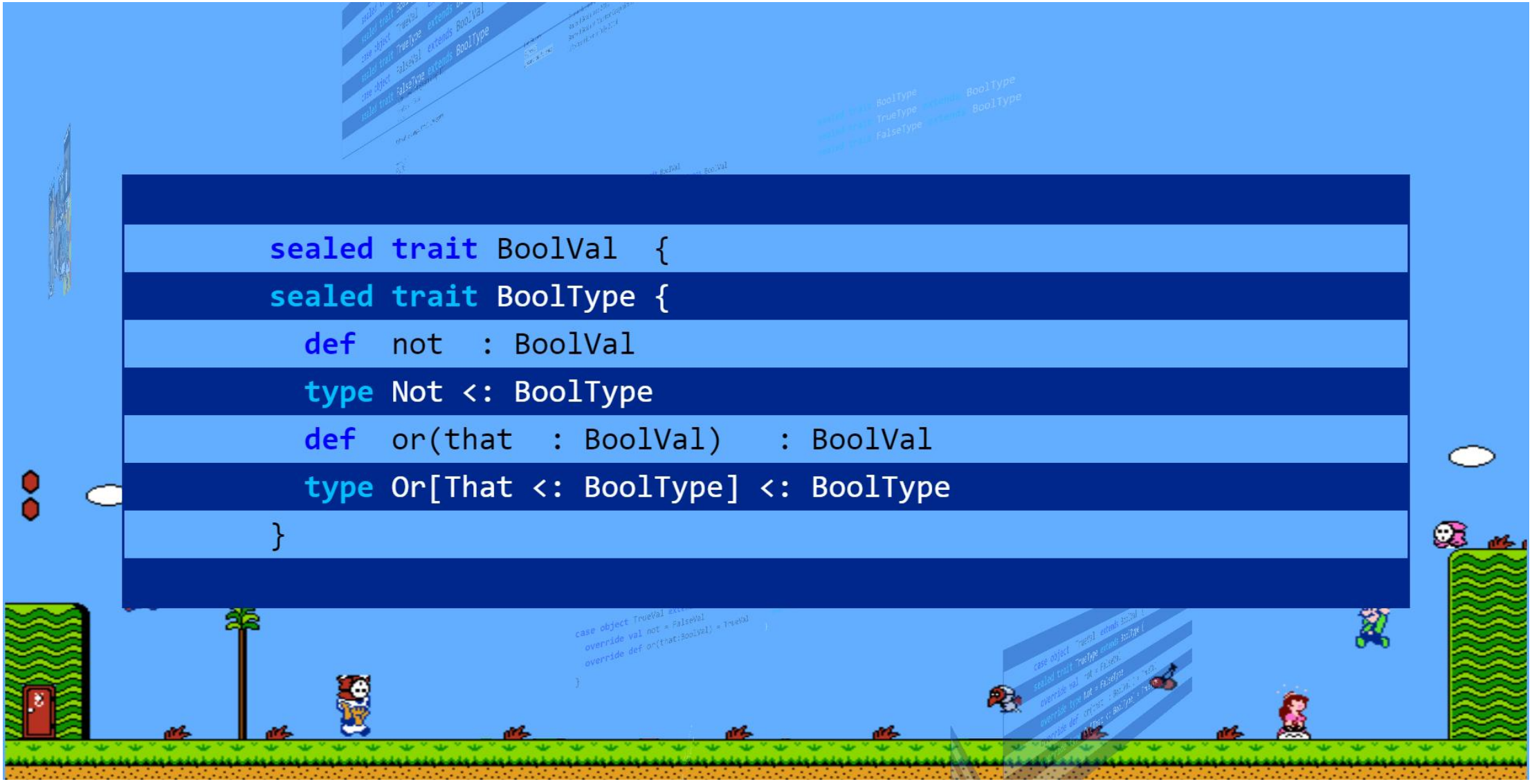
Primarily Java from 2004-2013
Started Scala late 2012
Started Scala at Mentor Graphics late 2013
Lift committer in July 2014

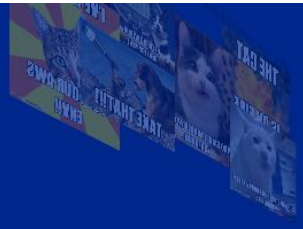
Programming in Scala is like...



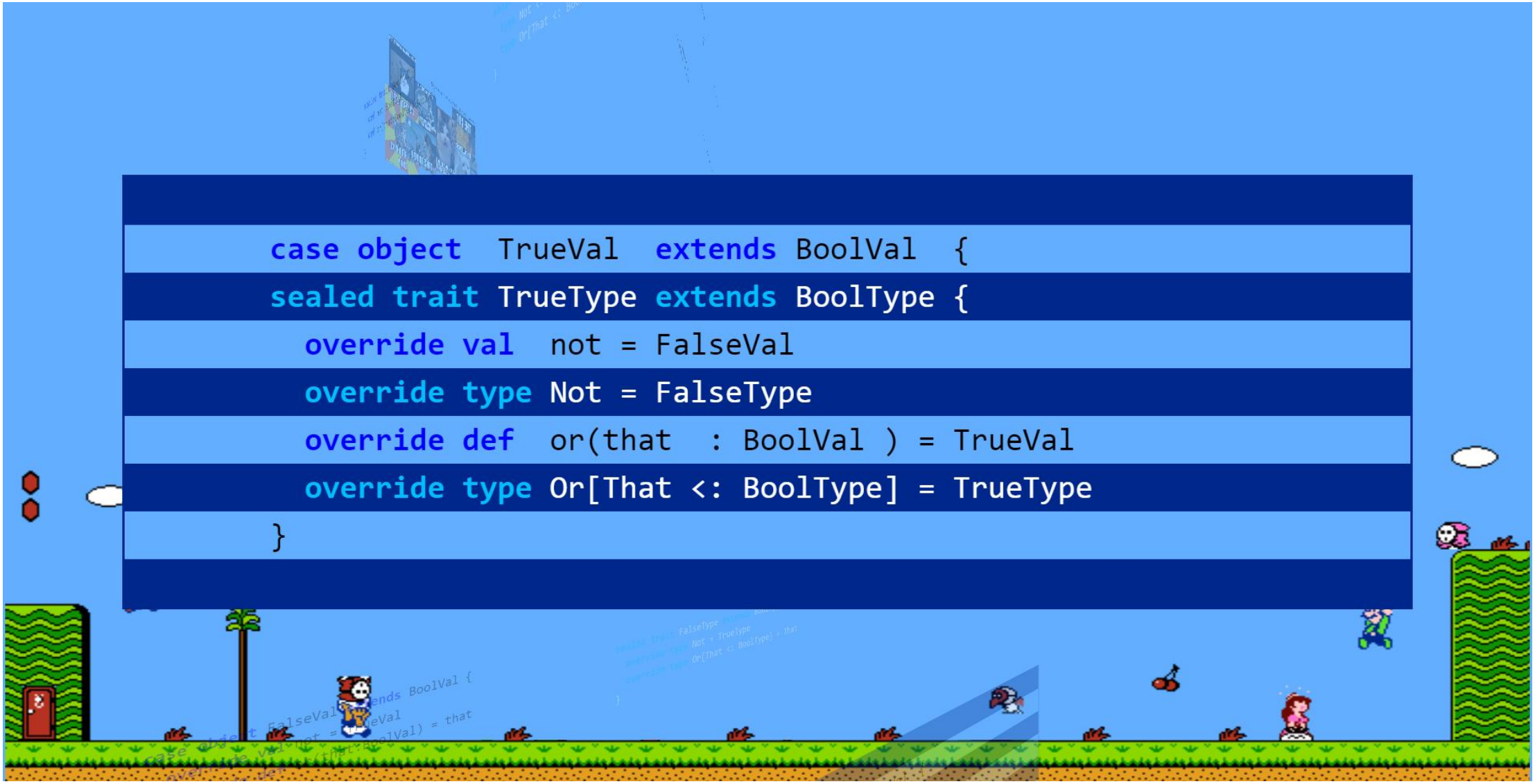

```
sealed trait BoolVal
sealed trait BoolType
case object  TrueVal   extends BoolVal
sealed trait TrueType  extends BoolType
case object  FalseVal  extends BoolVal
sealed trait FalseType extends BoolType
```


sealed trait BoolType {
type Not <: BoolType
type Or[That <: BoolType] <: BoolType
}

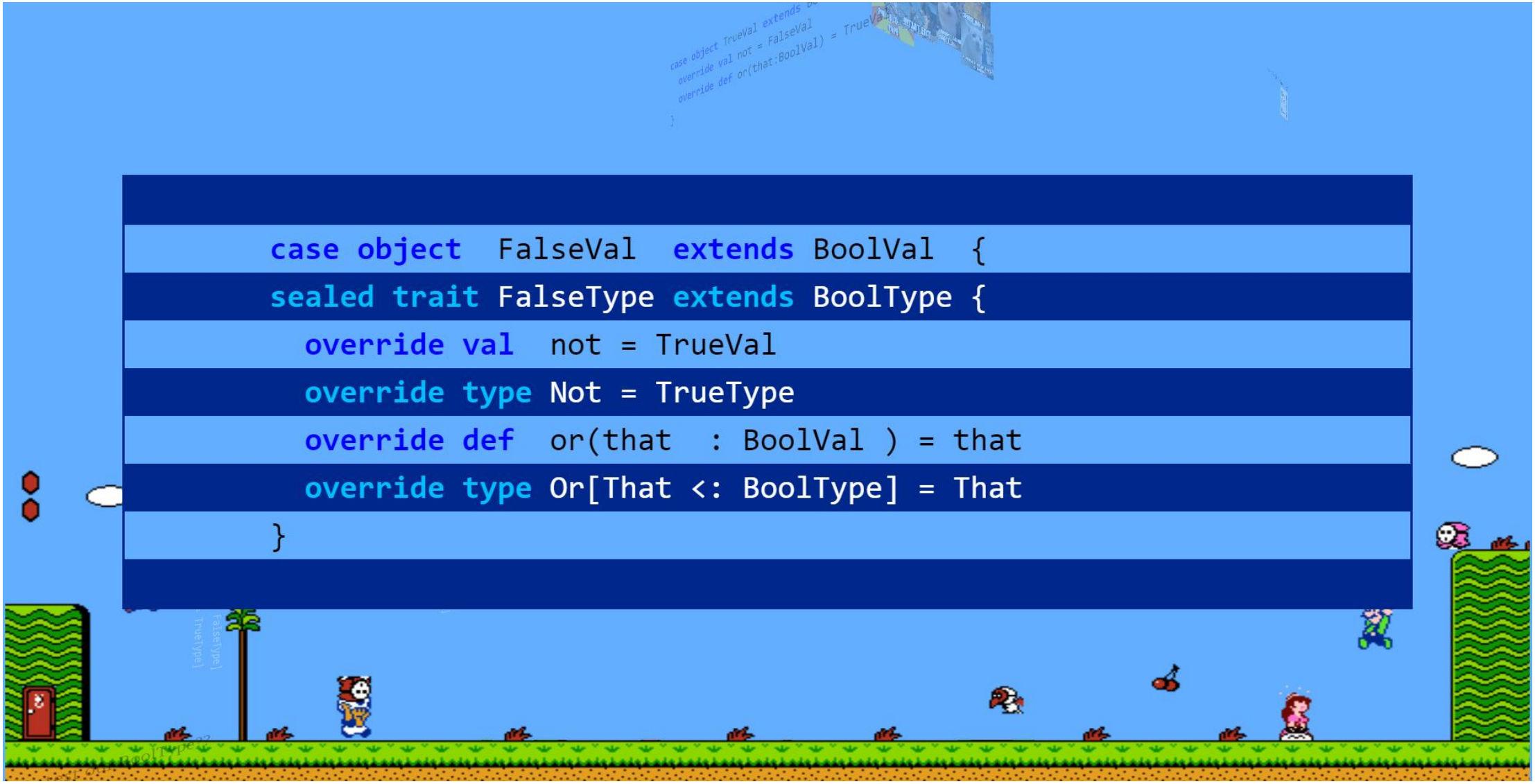

sealed trait BoolVal {
sealed trait BoolType {
def not : BoolVal
type Not <: BoolType
def or(that : BoolVal)   : BoolVal
type Or[That <: BoolType] <: BoolType
}



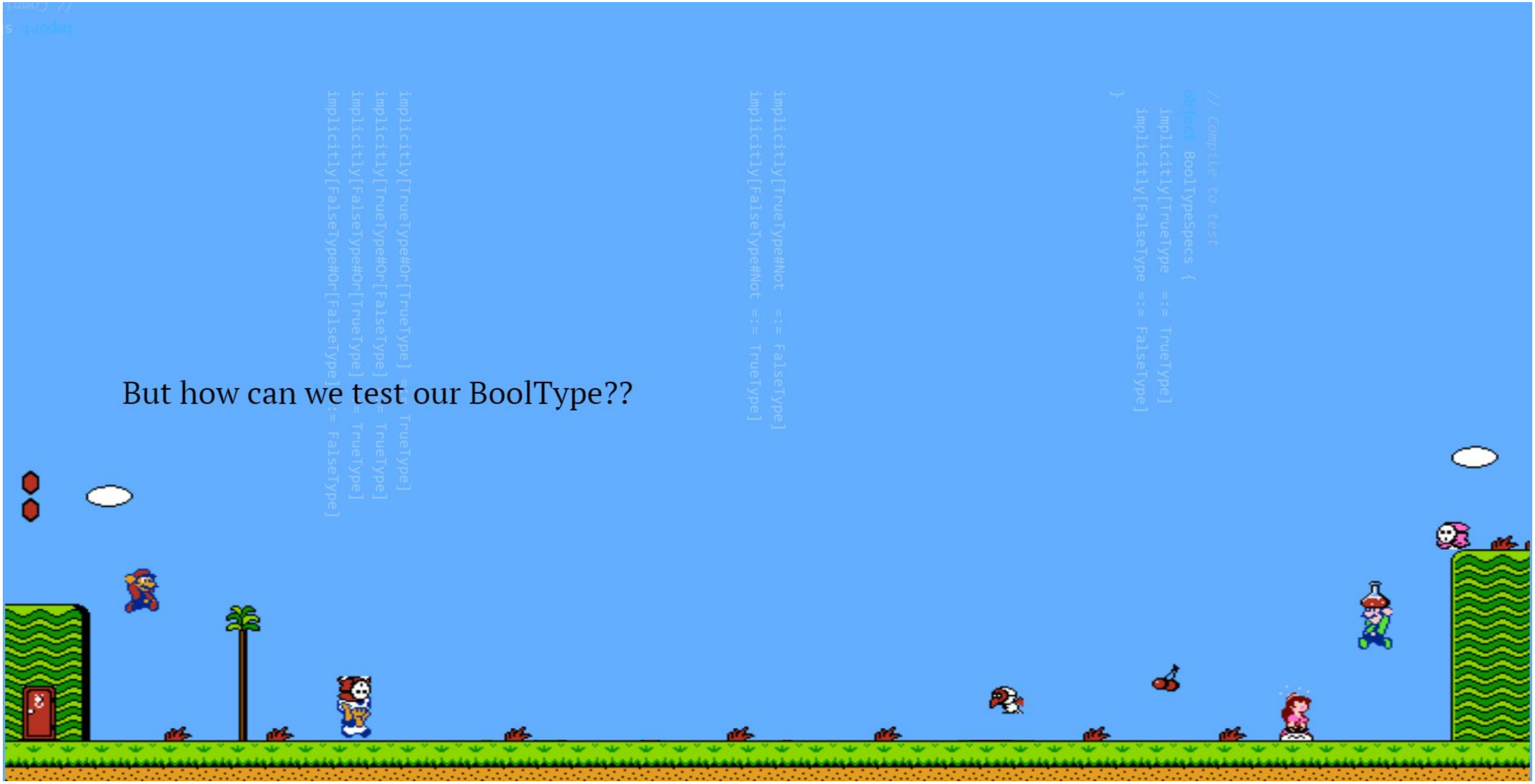
```scala
sealed trait TrueType extends BoolType {
  override type Not = FalseType
  override type Or[That <: BoolType] = TrueType
}
```


case object TrueVal extends BoolVal {
sealed trait TrueType extends BoolType {
override val not = FalseVal
override type Not = FalseType
override def or(that : BoolVal) = TrueVal
override type Or[That <: BoolType] = TrueType
}

```scala
sealed trait FalseType extends BoolType {
  override type Not = TrueType
  override type Or[That <: BoolType] = That
}
```

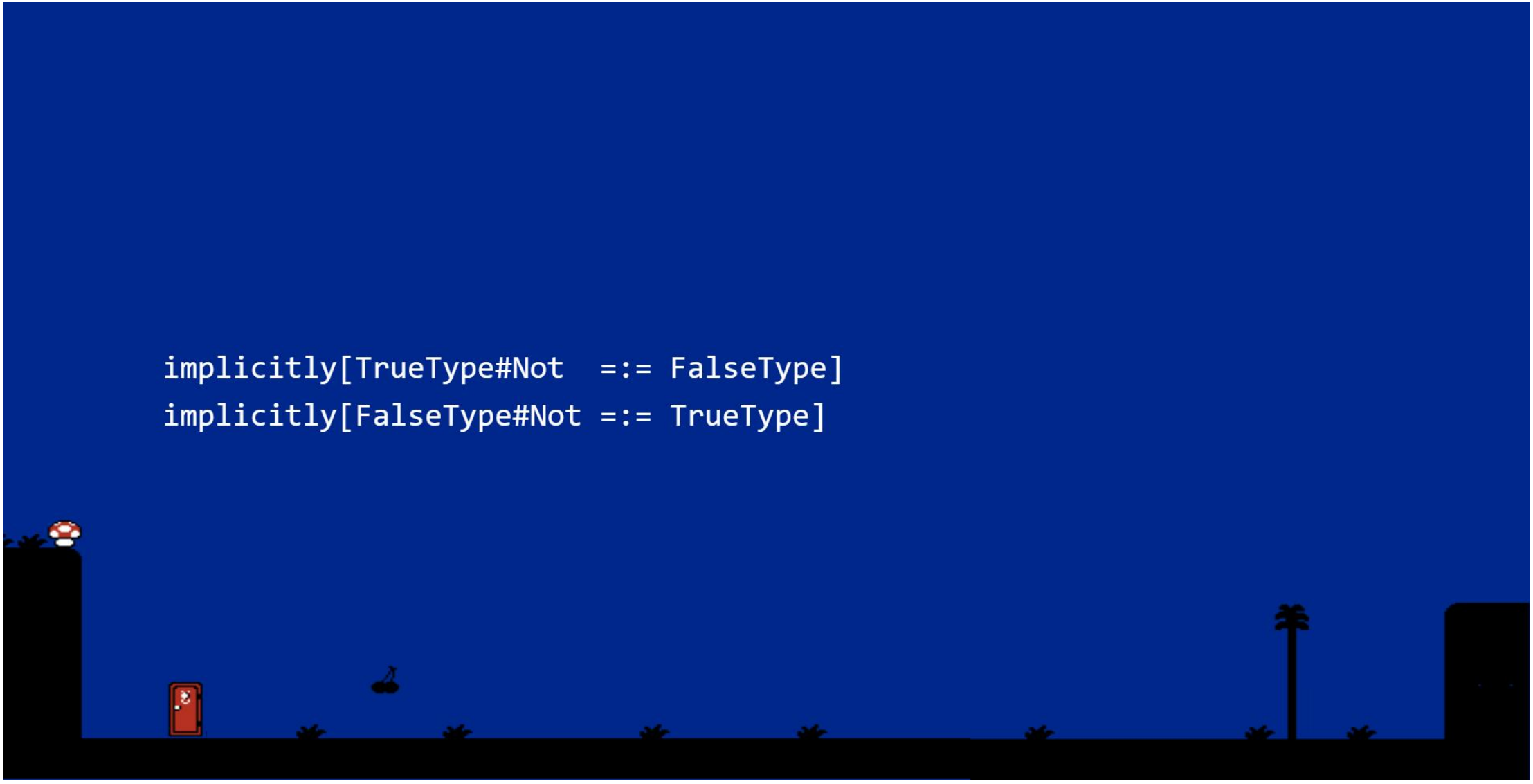

case object FalseVal extends BoolVal {
sealed trait FalseType extends BoolType {
override val not = TrueVal
override type Not = TrueType
override def or(that : BoolVal) = that
override type Or[That <: BoolType] = That
}

But how can we test our BoolType??



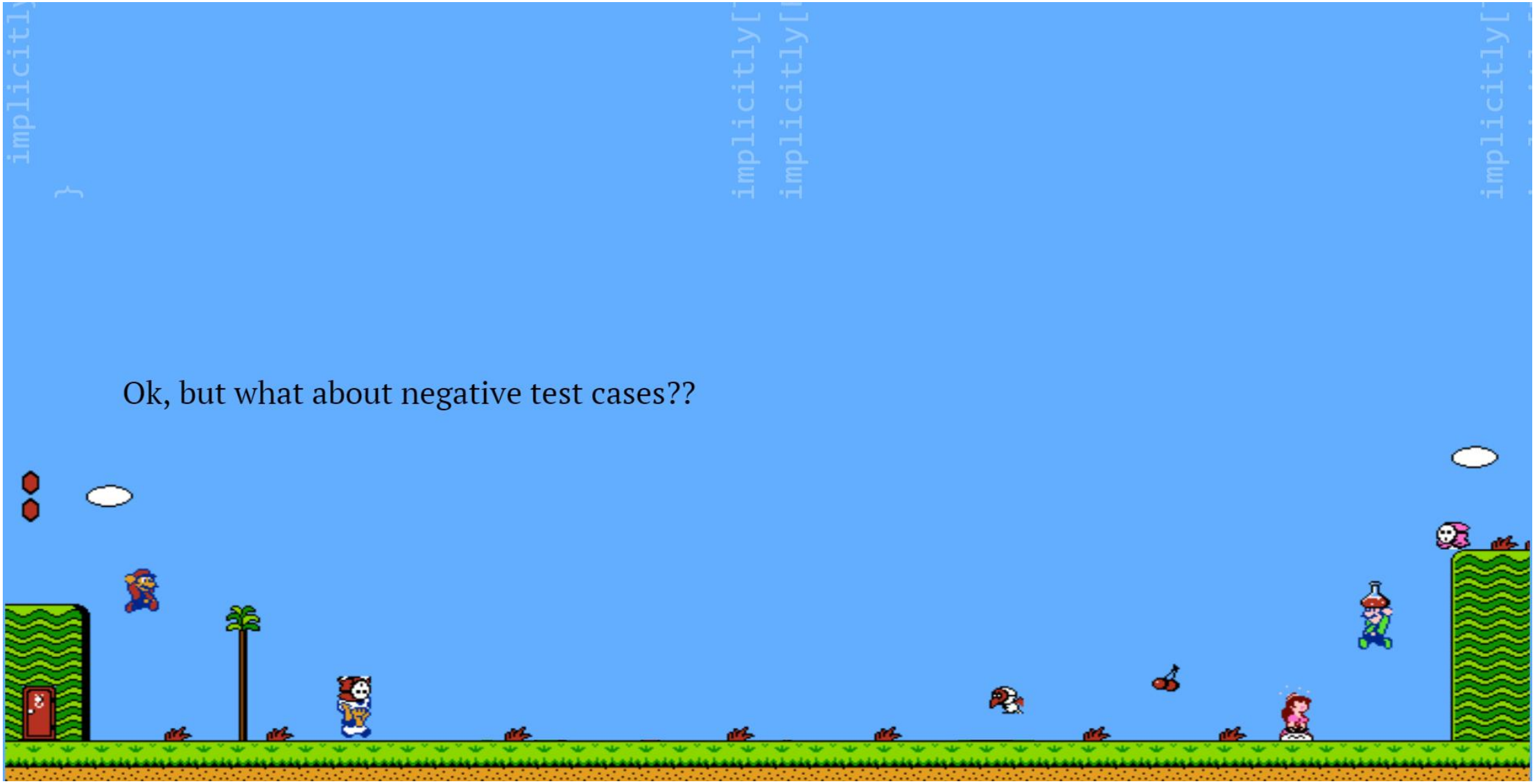
```scala
// Compile to test
object BoolTypeSpecs {
  implicitly[TrueType  =:= TrueType]
  implicitly[FalseType =:= FalseType]
}
```

```
implicitly[TrueType#Not  =:= FalseType]
implicitly[FalseType#Not =:= TrueType]
```



```
implicitly[TrueType#Or[TrueType] =:= TrueType]
implicitly[TrueType#Or[FalseType] =:= TrueType]
implicitly[FalseType#Or[TrueType] =:= TrueType]
implicitly[FalseType#Or[FalseType] =:= FalseType]
```

Ok, but what about negative test cases??

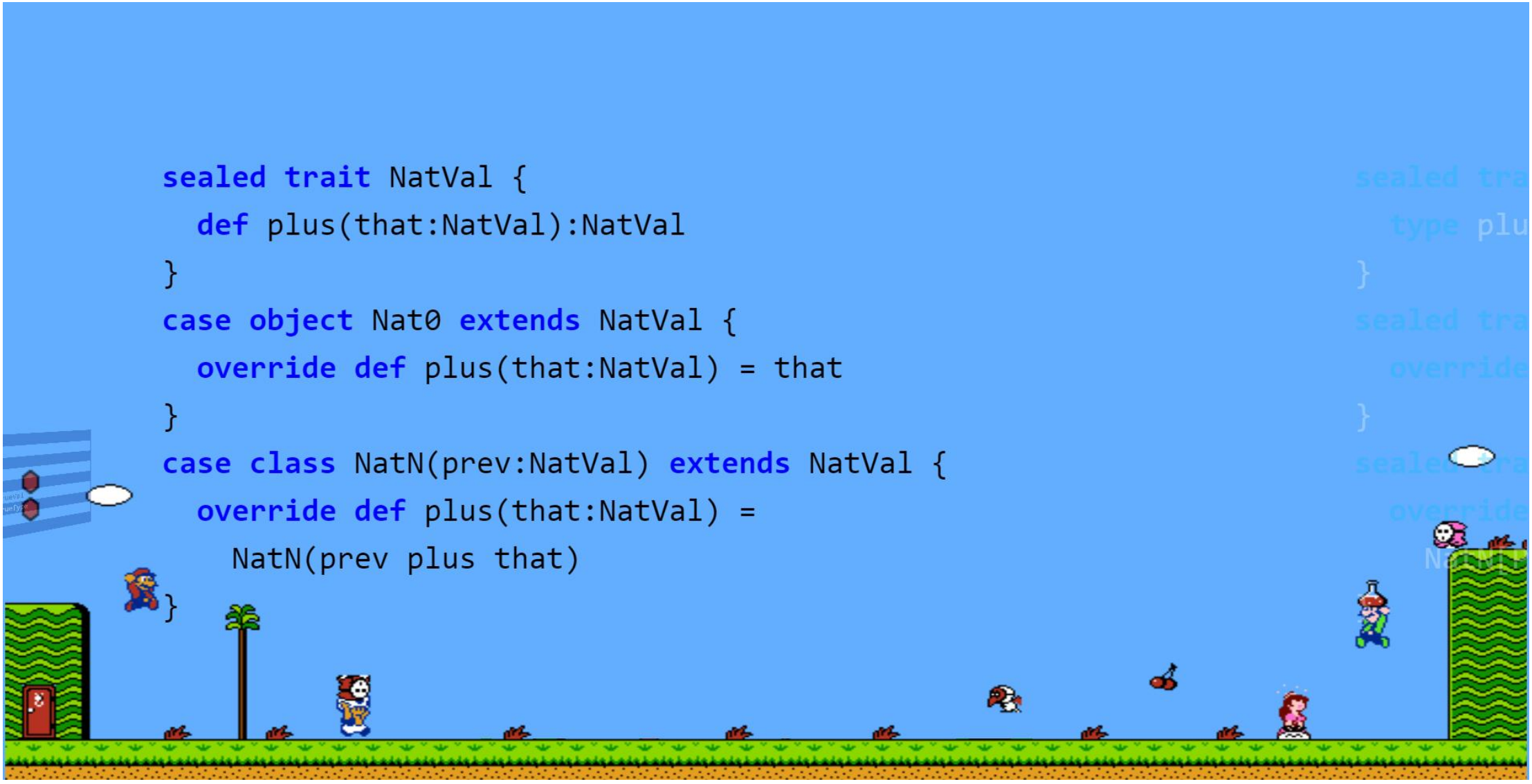


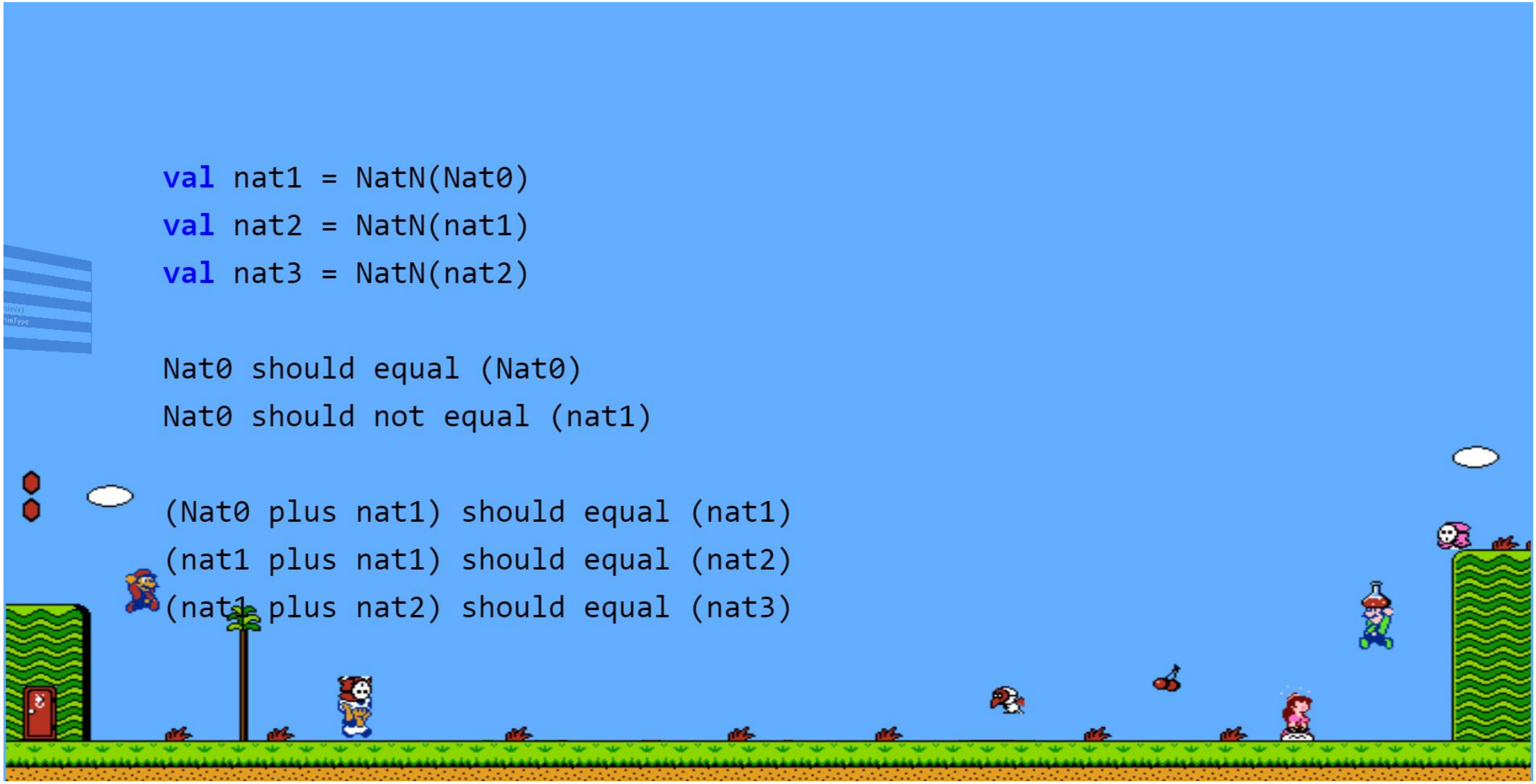
```scala
import shapeless.test.illTyped
// Compiles only if string DOESN'T compile
illTyped("implicitly[TrueType  =:= FalseType]")
illTyped("implicitly[FalseType =:= TrueType]")
```

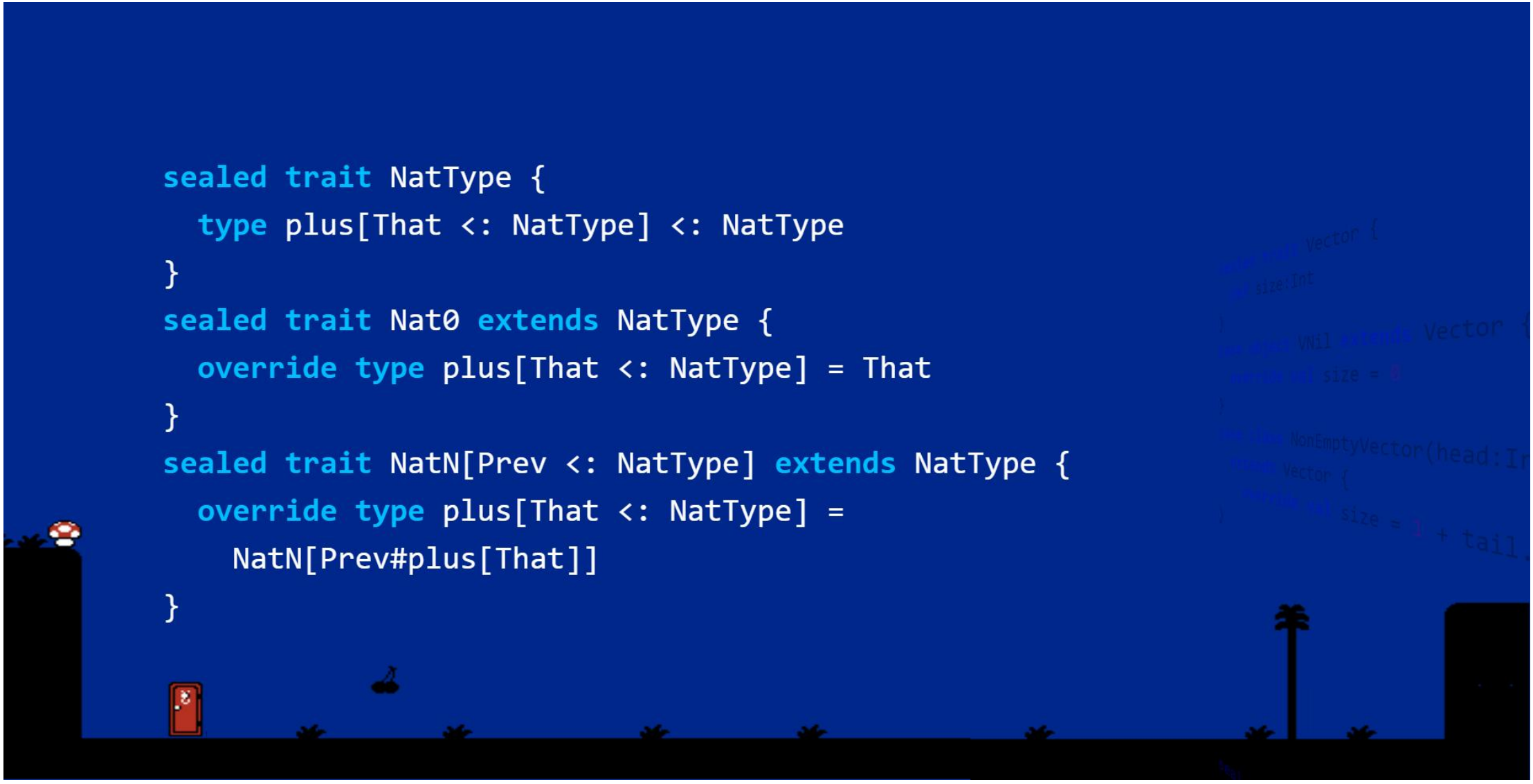
```
illTyped("implicitly[TrueType#Not  =:= TrueType]")
illTyped("implicitly[FalseType#Not =:= FalseType]")
illTyped("implicitly[TrueType#Or[TrueType] =:= TrueType]")
illTyped("implicitly[TrueType#Or[FalseType] =:= TrueType]")
```

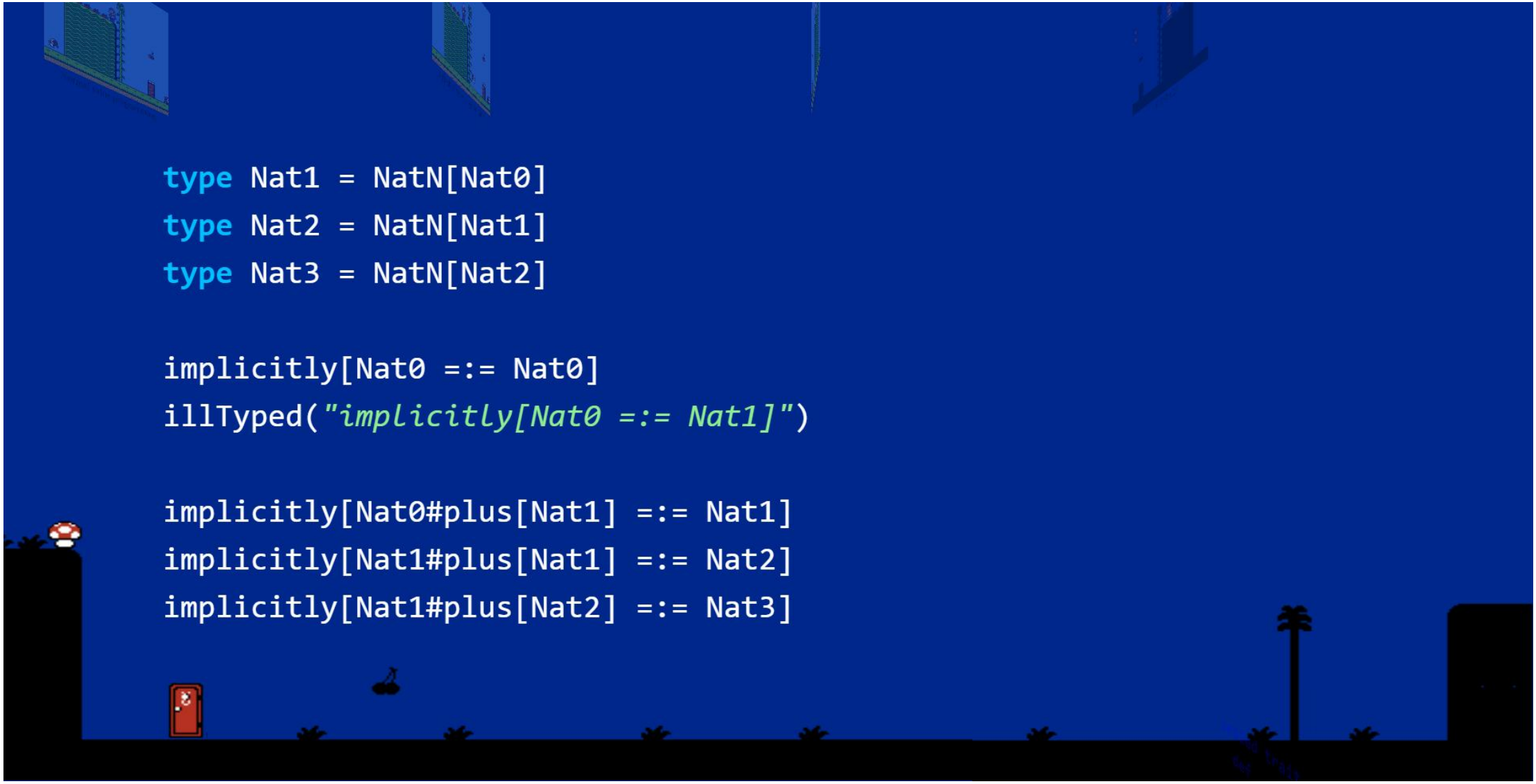

MARIO MADNESS
Nintendo
type Nat1
type Nat2
type Nat3
implicitly
illTyped("
implicitly
Well that was easy... It's just two types, really...

```scala
sealed trait NatVal {
  def plus(that:NatVal):NatVal
}
case object Nat0 extends NatVal {
  override def plus(that:NatVal) = that
}
case class NatN(prev:NatVal) extends NatVal {
  override def plus(that:NatVal) =
    NatN(prev plus that)
}
```




val nat1 = NatN(Nat0)
val nat2 = NatN(nat1)
val nat3 = NatN(nat2)
Nat0 should equal (Nat0)
Nat0 should not equal (nat1)
(Nat0 plus nat1) should equal (nat1)
(nat1 plus nat1) should equal (nat2)
(nat1 plus nat2) should equal (nat3)

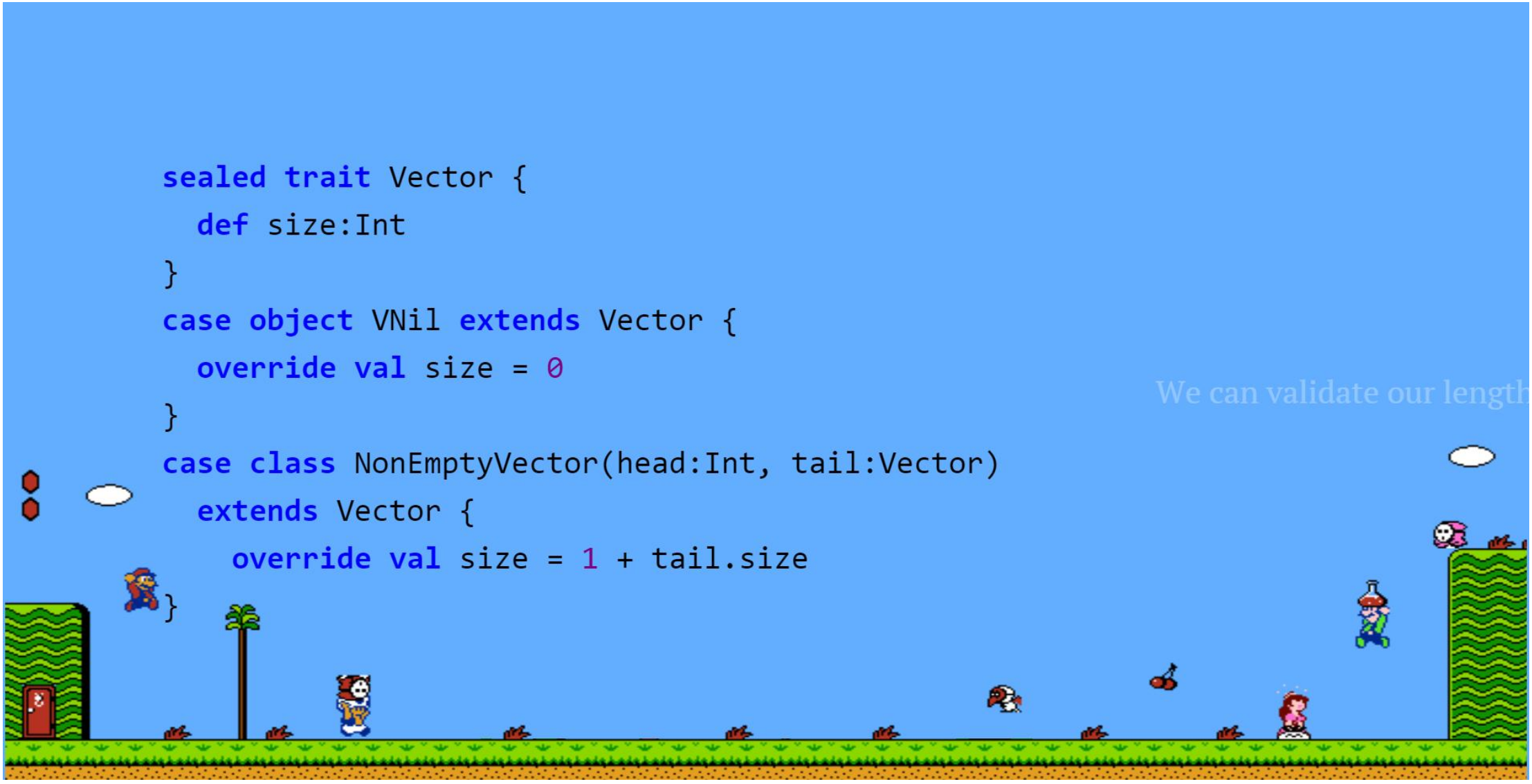

sealed trait NatType {
type plus[That <: NatType] <: NatType
}
sealed trait Nat0 extends NatType {
override type plus[That <: NatType] = That
}
sealed trait NatN[Prev <: NatType] extends NatType {
override type plus[That <: NatType] =
NatN[Prev#plus[That]]
}


type Nat1 = NatN[Nat0]
type Nat2 = NatN[Nat1]
type Nat3 = NatN[Nat2]
implicitly[Nat0 =:= Nat0]
illTyped("implicitly[Nat0 =:= Nat1]")
implicitly[Nat0#plus[Nat1] =:= Nat1]
implicitly[Nat1#plus[Nat1] =:= Nat2]
implicitly[Nat1#plus[Nat2] =:= Nat3]


Types, types, and more types, blah, blah, blah. What good is it??

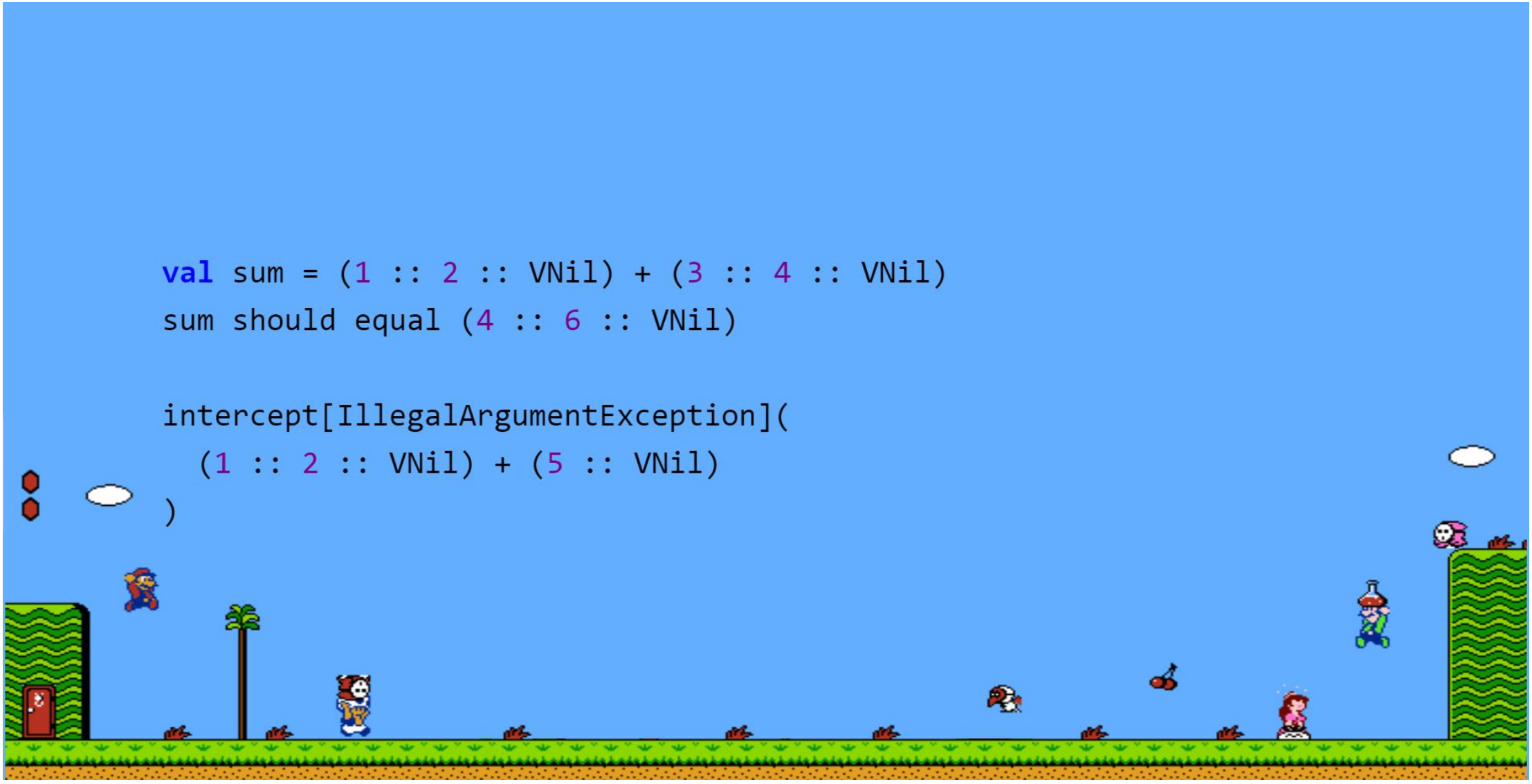
```scala
sealed trait Vector {
  def size:Int
}
case object VNil extends Vector {
  override val size = 0
}
case class NonEmptyVector(head:Int, tail:Vector)
  extends Vector {
    override val size = 1 + tail.size
}
```

We can validate our length

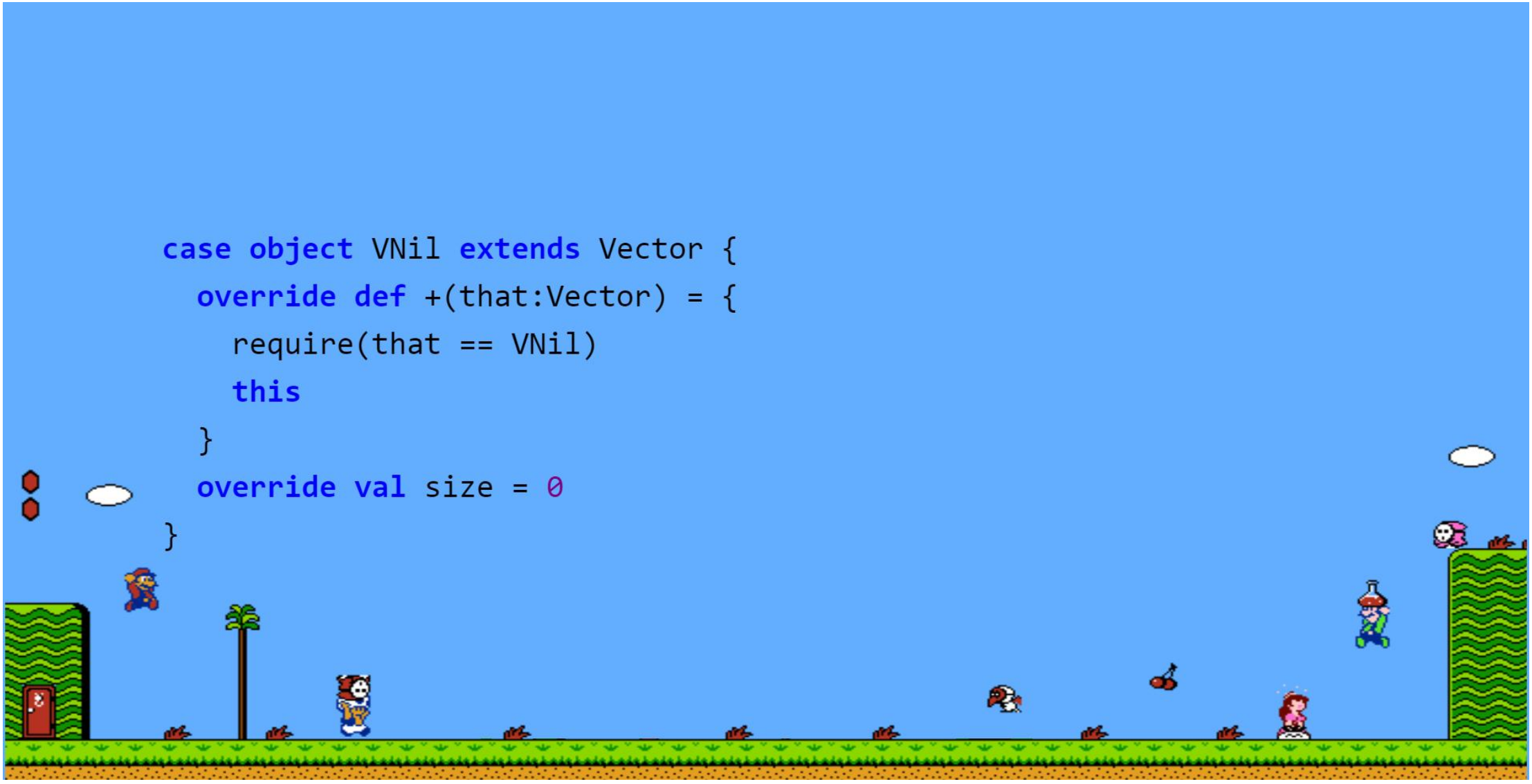


```
sealed trait Vector {
  // Convenience vector constructor
  def ::(head:Int):Vector = NonEmptyVector(head, this)
  // Vector addition
  def +(that:Vector):Vector
  def size:Int
}
```

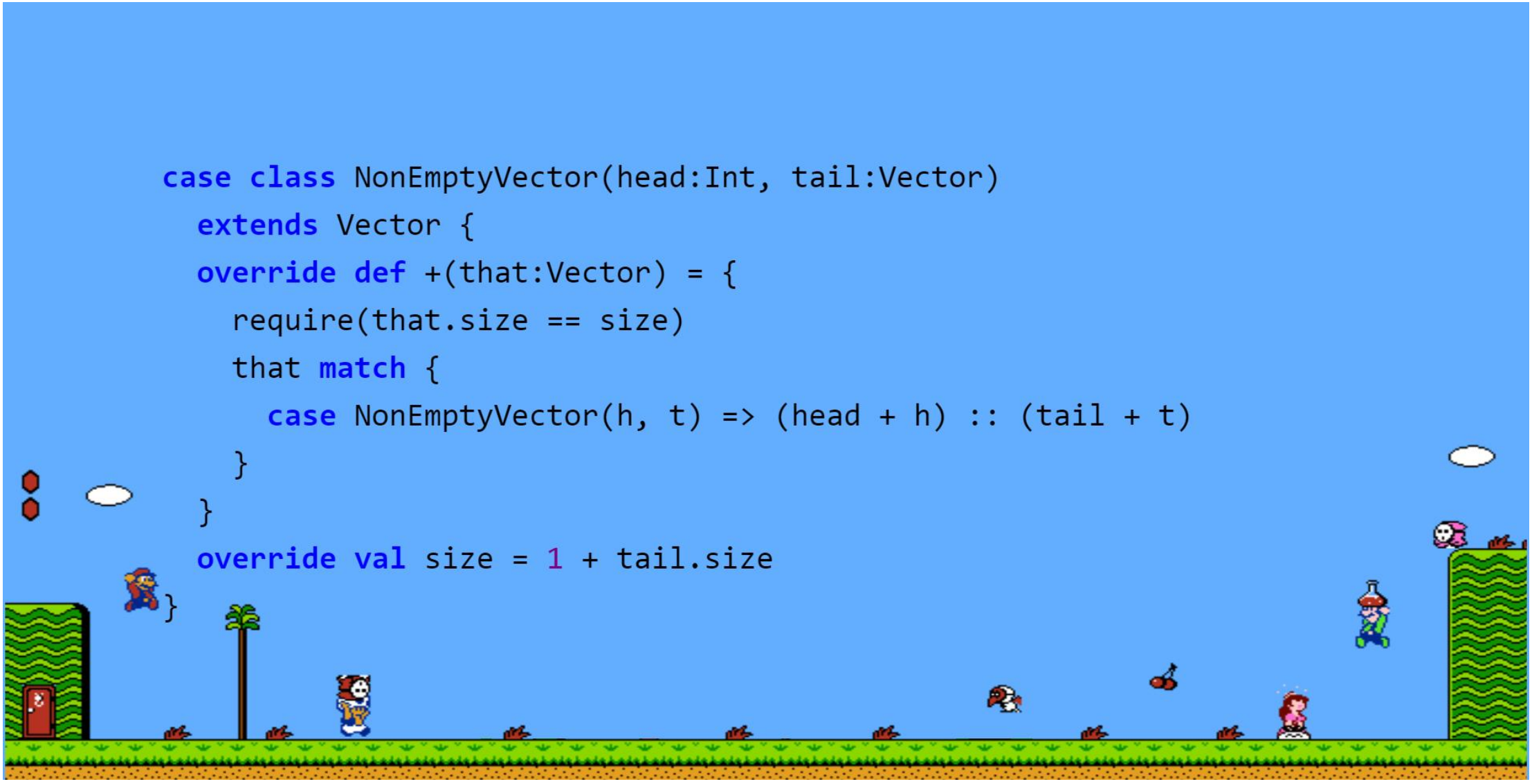
```
val sum = (1 :: 2 :: VNil) + (3 :: 4 :: VNil)
sum should equal (4 :: 6 :: VNil)

intercept[IllegalArgumentException](
  (1 :: 2 :: VNil) + (5 :: VNil)
)
```



```
case object VNil extends Vector {
  override def +(that:Vector) = {
    require(that == VNil)
    this
  }
  override val size = 0
}
```



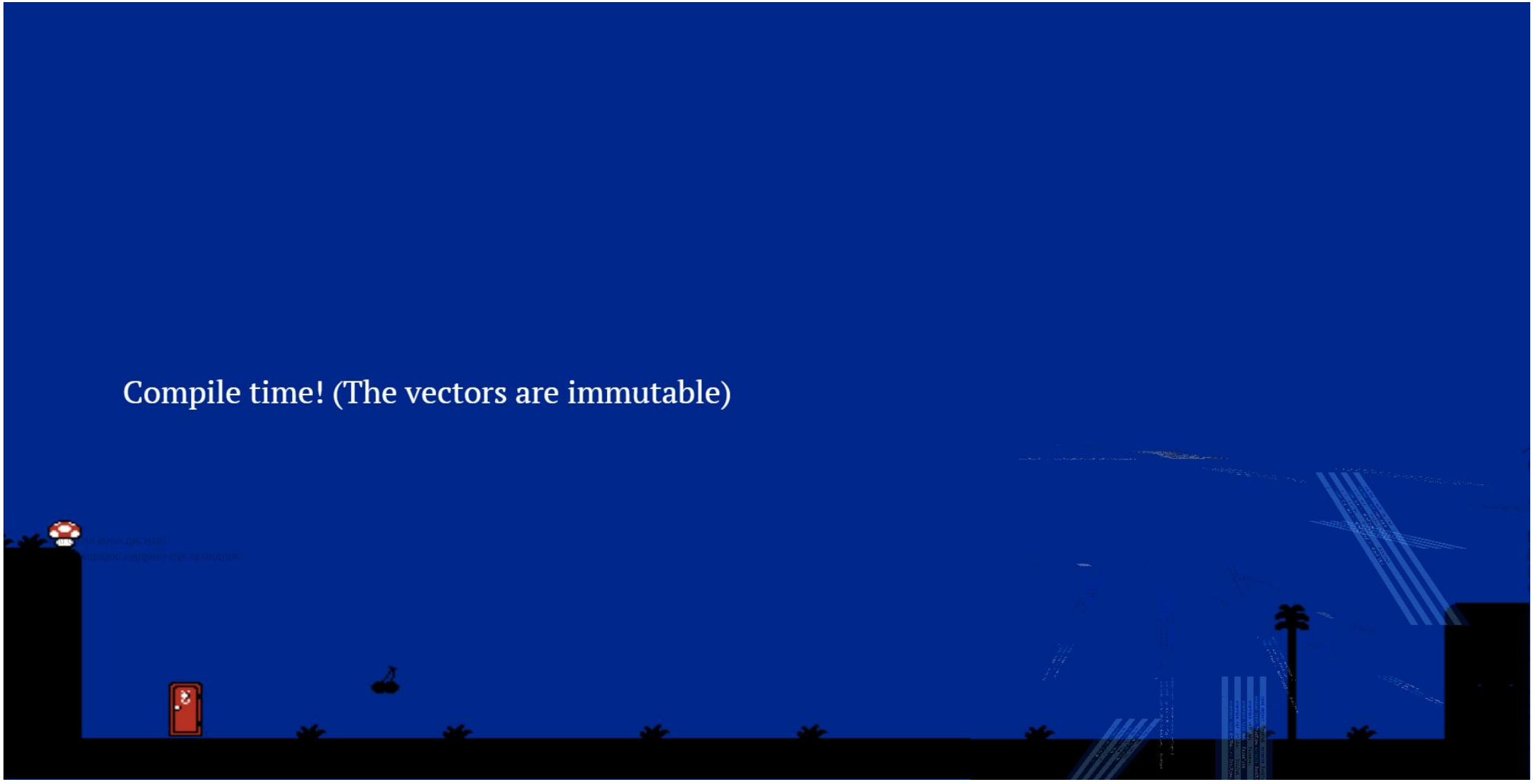
```
case class NonEmptyVector(head:Int, tail:Vector)
  extends Vector {
  override def +(that:Vector) = {
    require(that.size == size)
    that match {
      case NonEmptyVector(h, t) => (head + h) :: (tail + t)
    }
  }
  override val size = 1 + tail.size
}
```



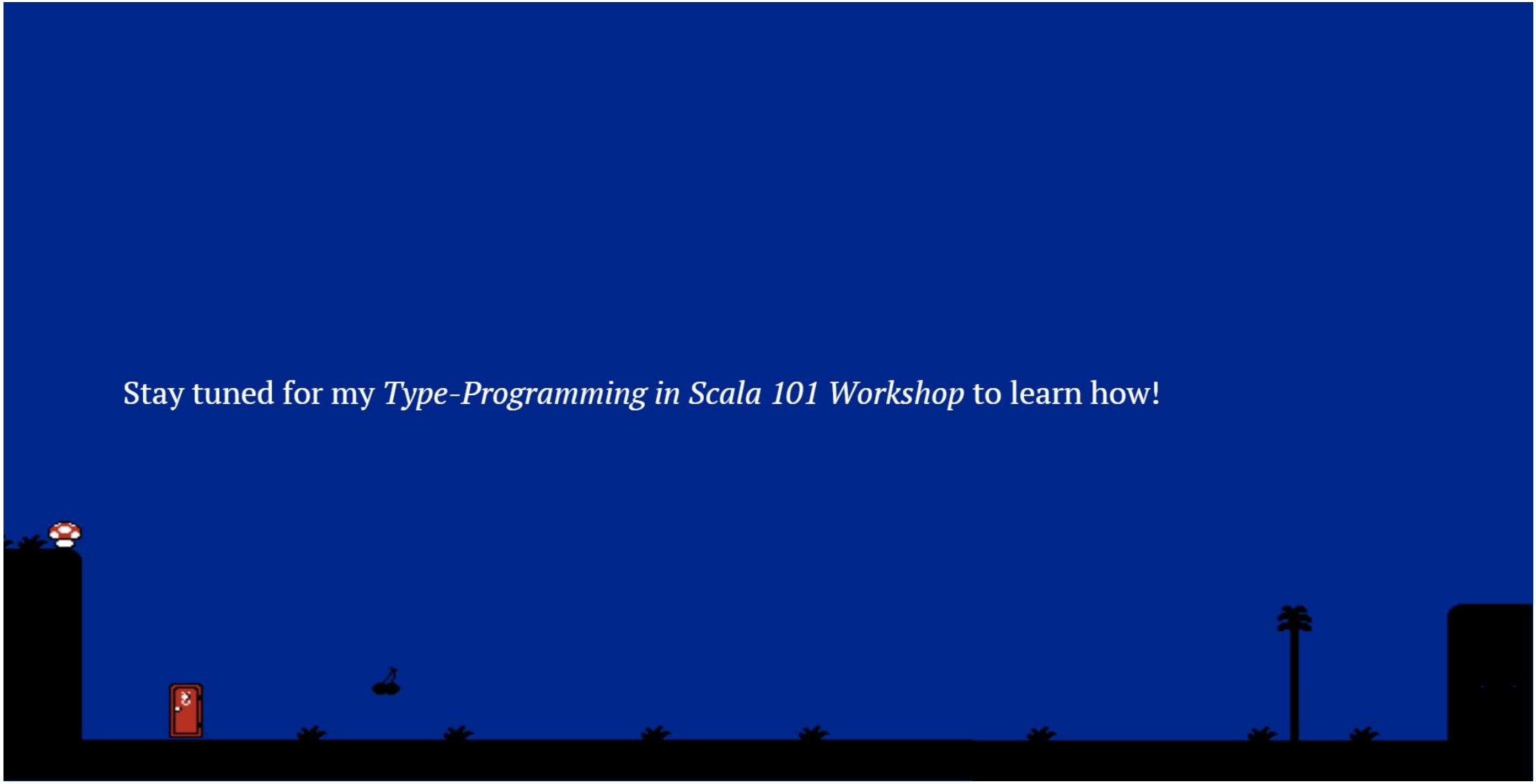
This implementation validates size at runtime.

But *when* do we know the size?

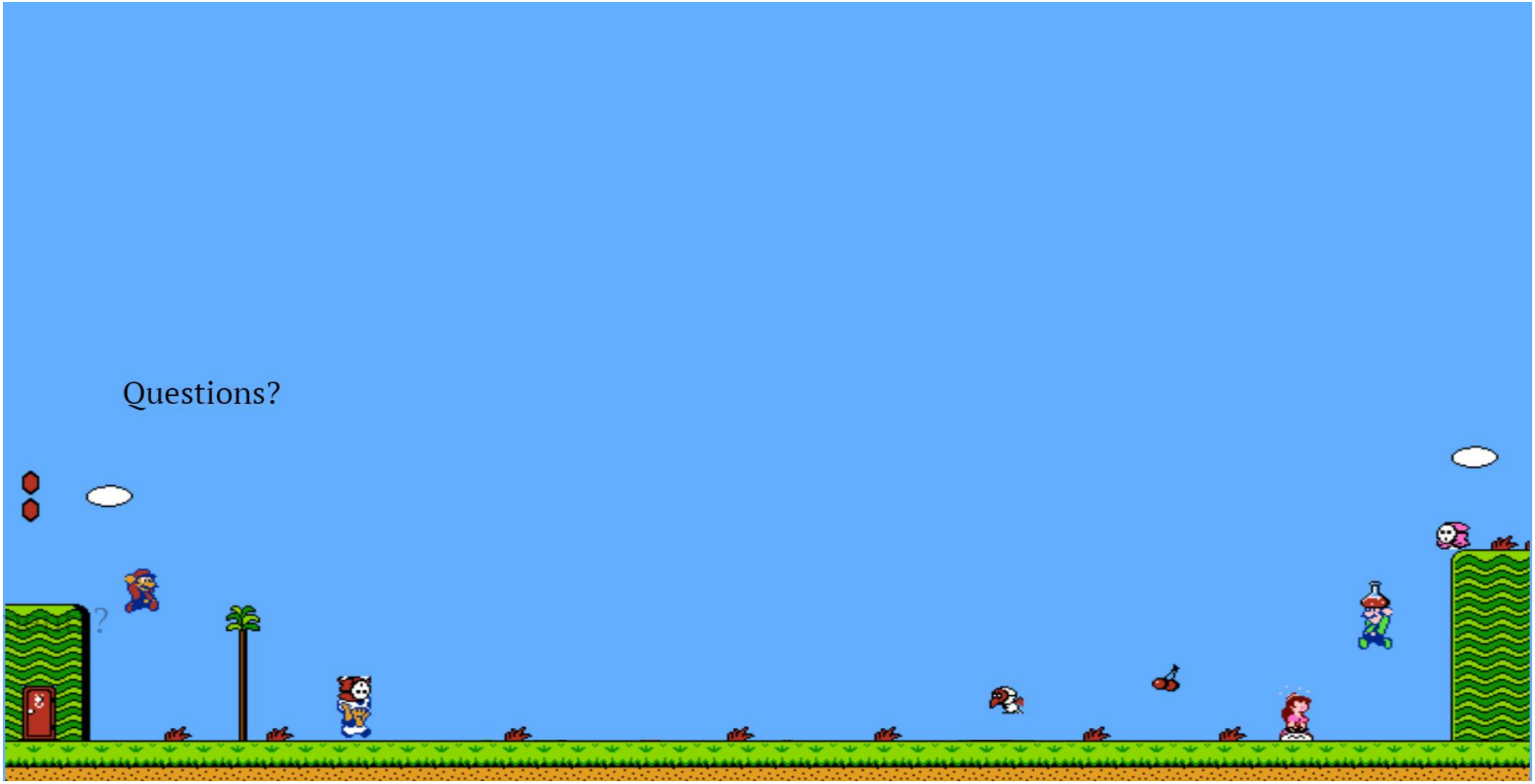
Compile time! (The vectors are immutable)




We can validate our lengths without a runtime IllegalArgumentException!


Stay tuned for my *Type-Programming in Scala 101 Workshop* to learn how!

Questions?



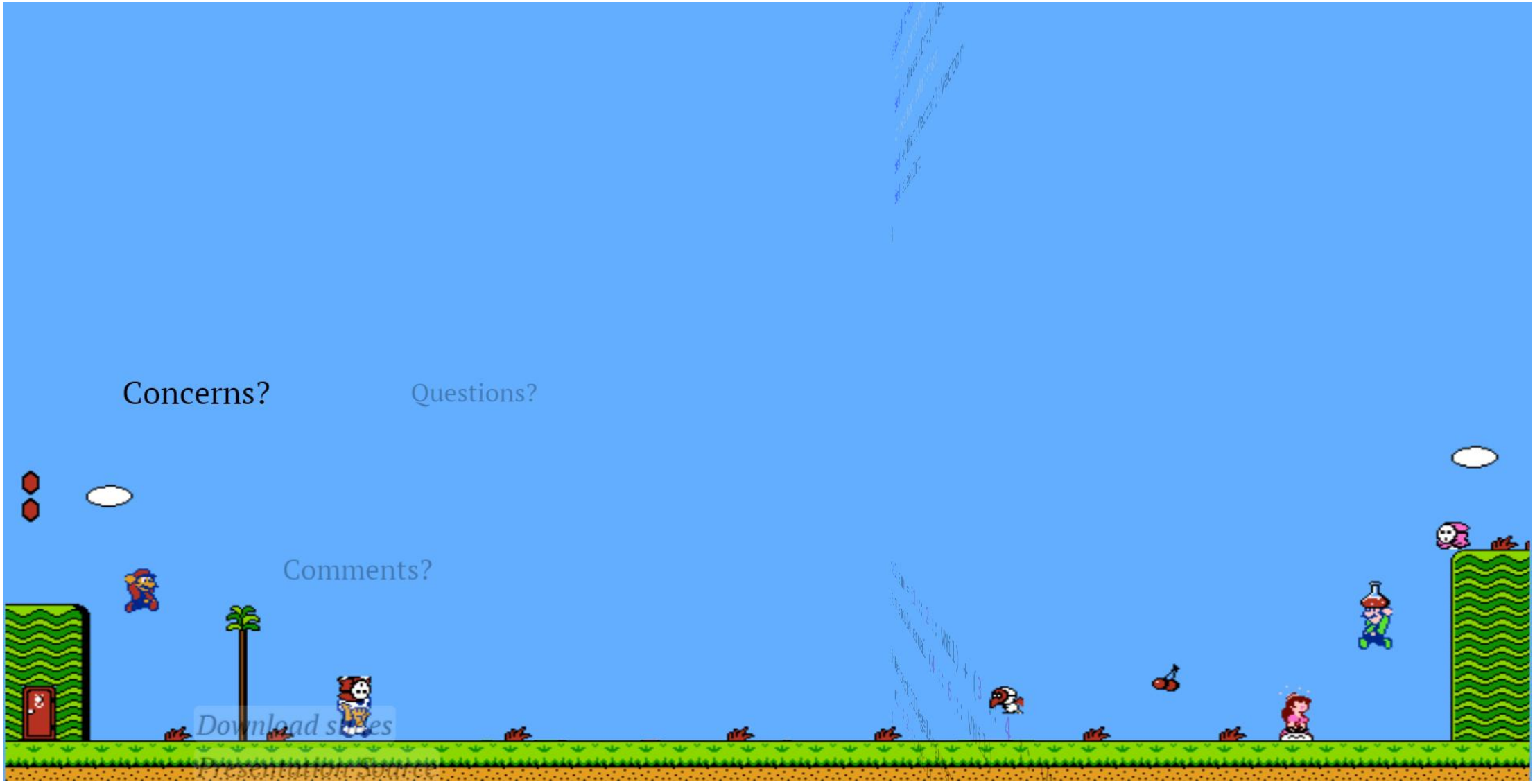
erns?

Questions?

Comments?

Download slides

Presentation Source


Concerns?
Questions?
Comments?
Download slides
Presentation Source

Questions?

Comments?

Download slides

Presentation Source