Type-Level Programming

The Subspace of Scala

Joe Barnes

Follow along at http://type-prog.herokuapp.com

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??

se

ca

ca

Not much…

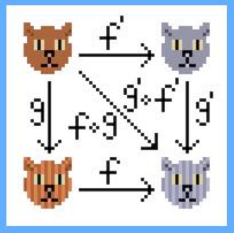
So… Type programming…

What do *you* know about *tha*

I've made zero contributions to shapeless

Not much...

Also no contributions to Cats



Type-Level Programming

The Subspace of Scala

Joe Barnes

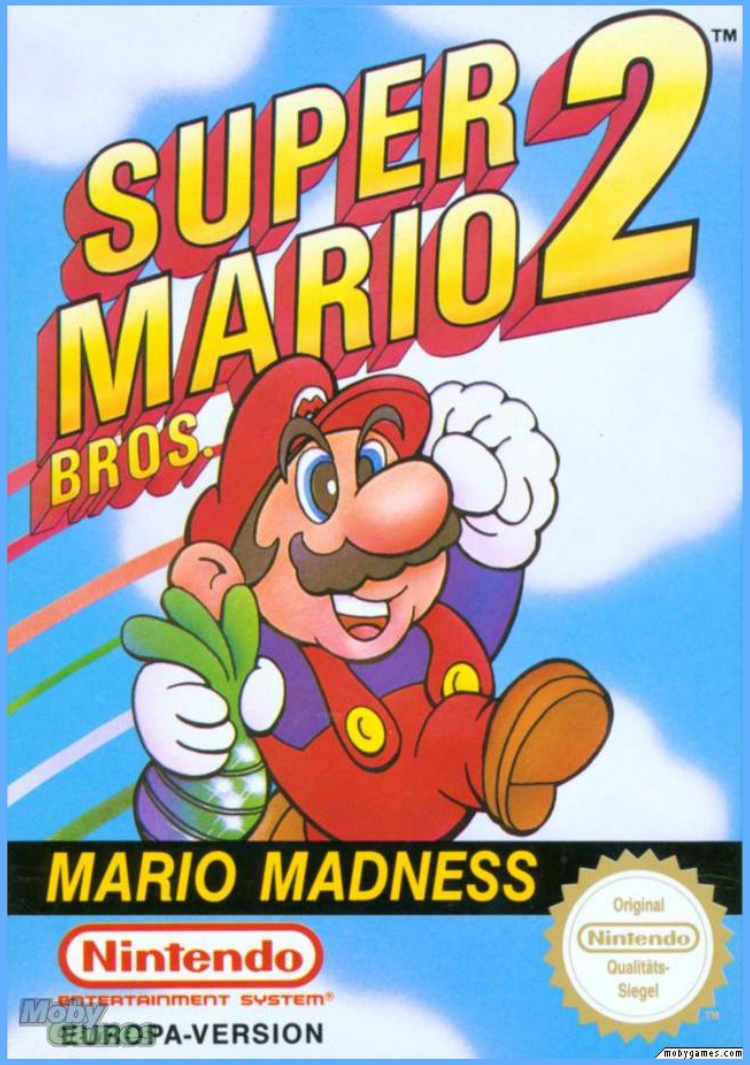
Follow along at http://type-prog.herokuapp.com

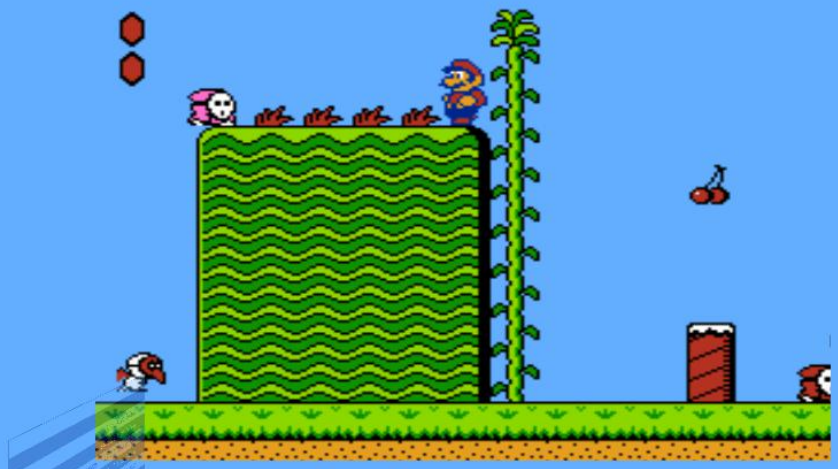
Powered by Lift

Although I have contributed plenty of cat memes!

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

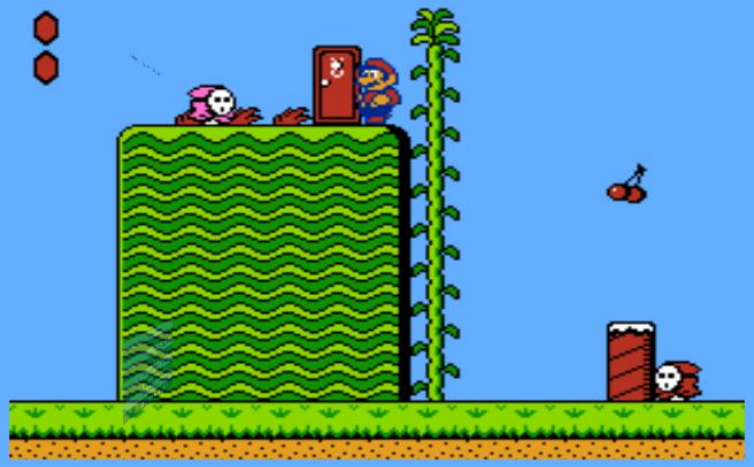
Programming in Scala is like...


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

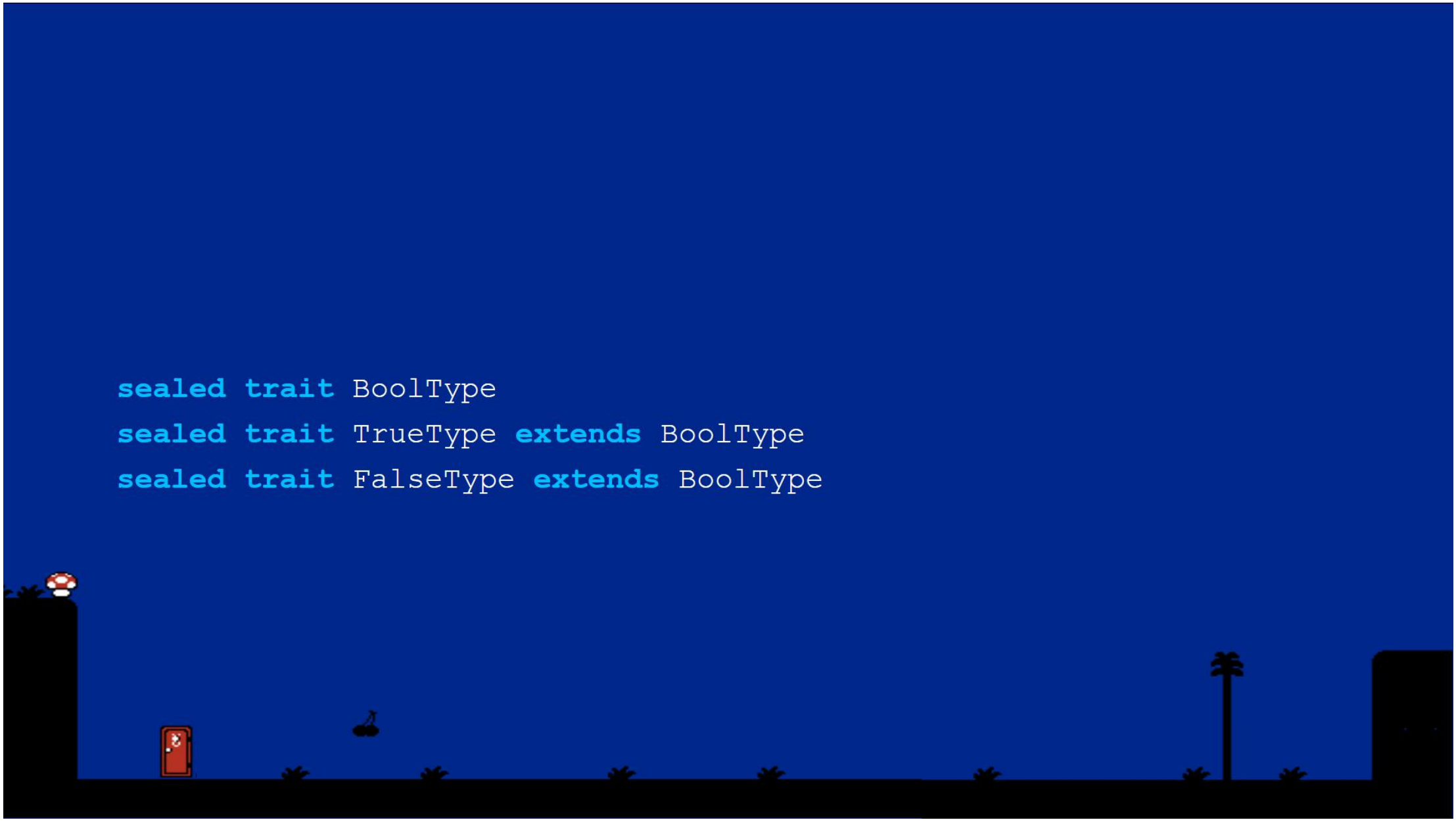
```
sealed trait BoolVal
case object TrueVal extends BoolVal
case object FalseVal extends BoolVal
```

```scala
sealed trait BoolVal {
  def not:BoolVal
  def or(that:BoolVal):BoolVal
}
```

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

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

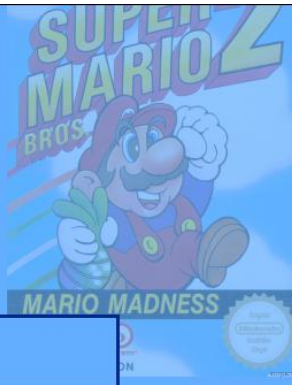
```
sealed trait BoolType
sealed trait TrueType extends BoolType
sealed trait FalseType extends BoolType
```



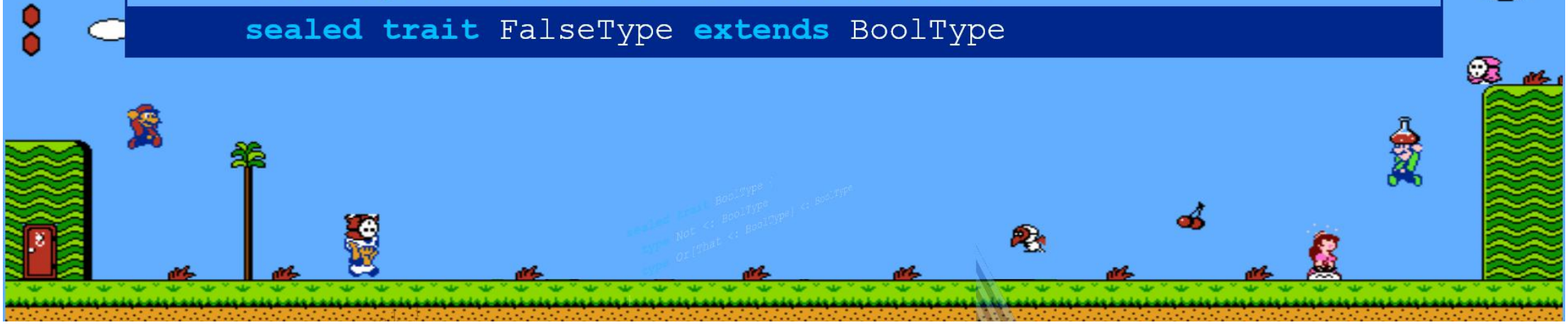
oe Barnes
@joescii
rose :: and :: conz

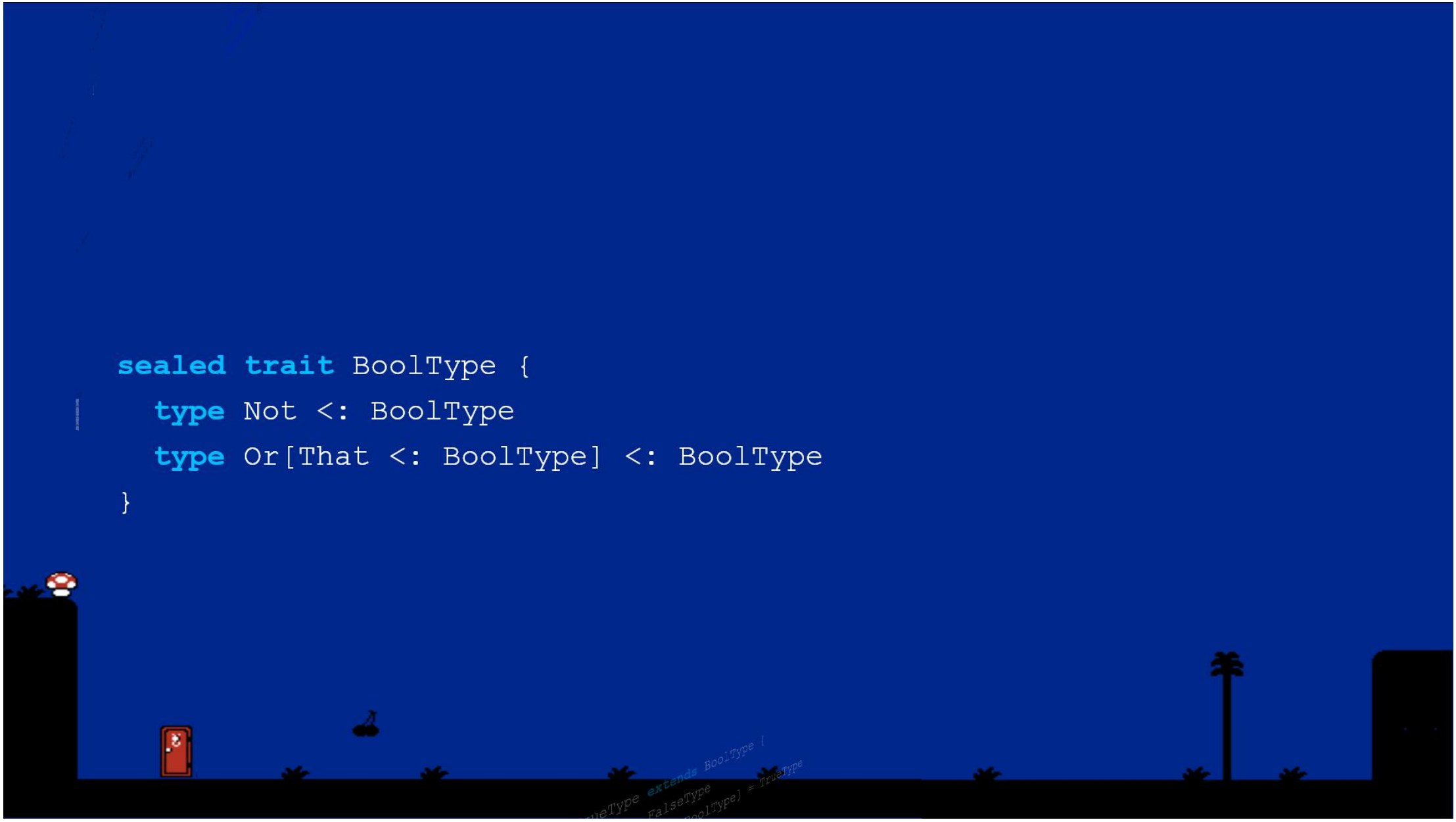
Primarily Java from 2004-2013
Started Scala late 2012
Started Scala at Mentor Graphics late 2013
Lift committer in July 2014

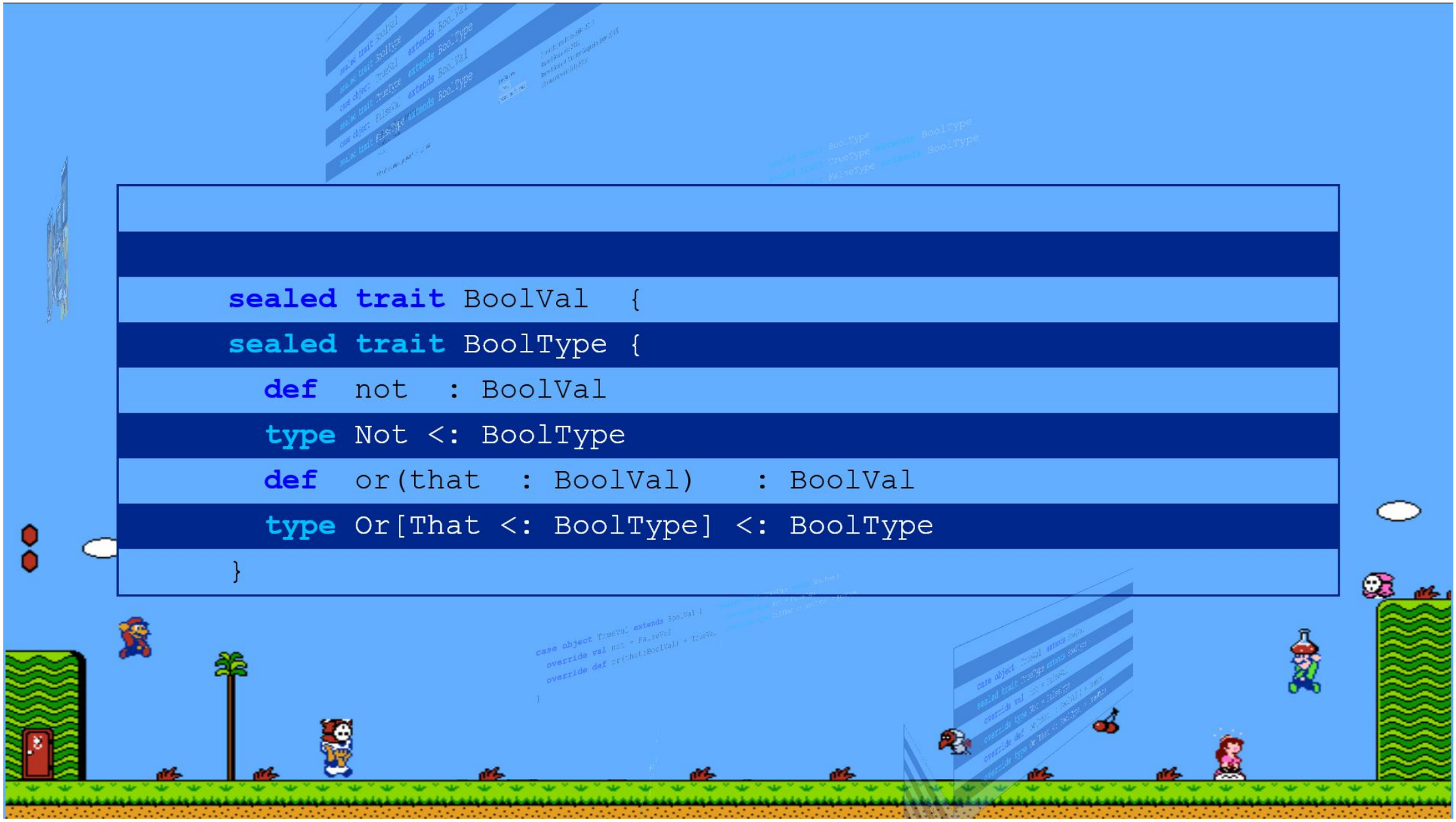
Programming in Scala is like...

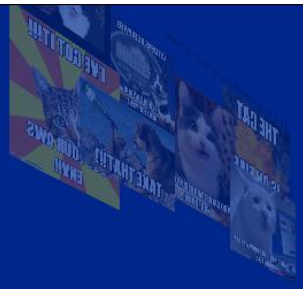



```
sealed trait BoolVal
sealed trait BoolType
case object  TrueVal   extends BoolVal
sealed trait TrueType  extends BoolType
case object  FalseVal  extends BoolVal
sealed trait FalseType extends BoolType
```

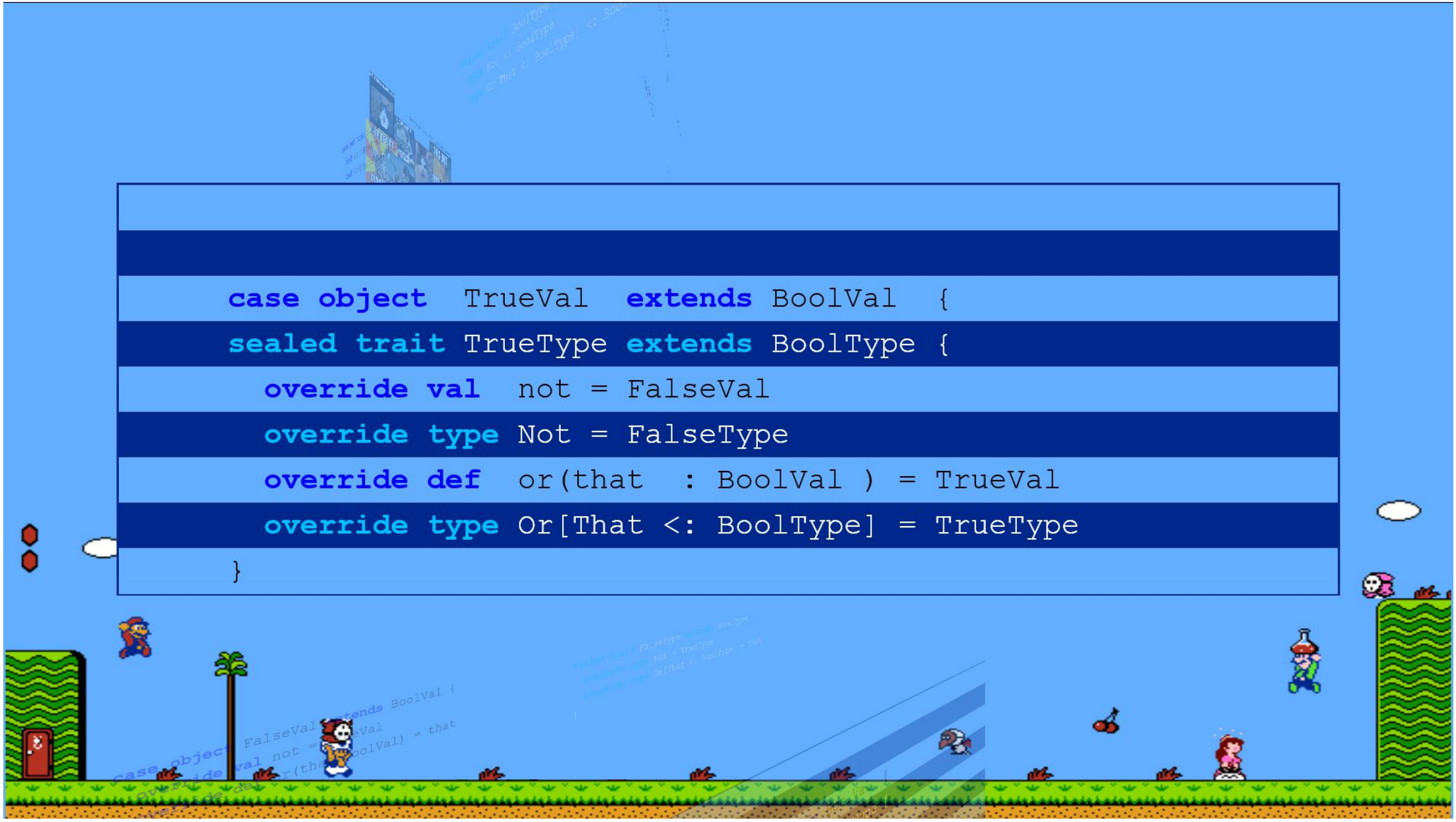


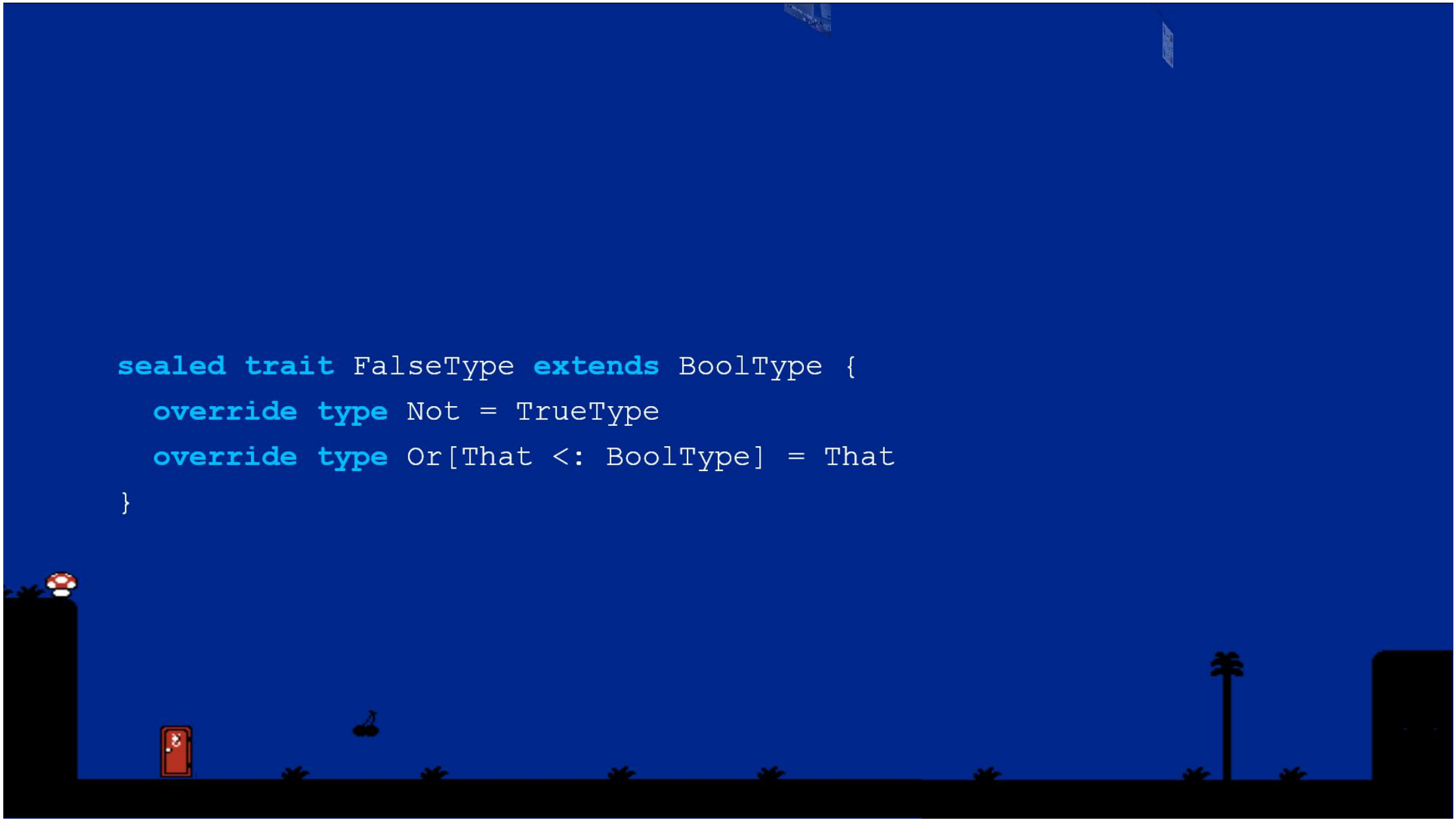

sealed trait BoolType {
type Not <: BoolType
type Or[That <: BoolType] <: BoolType
}

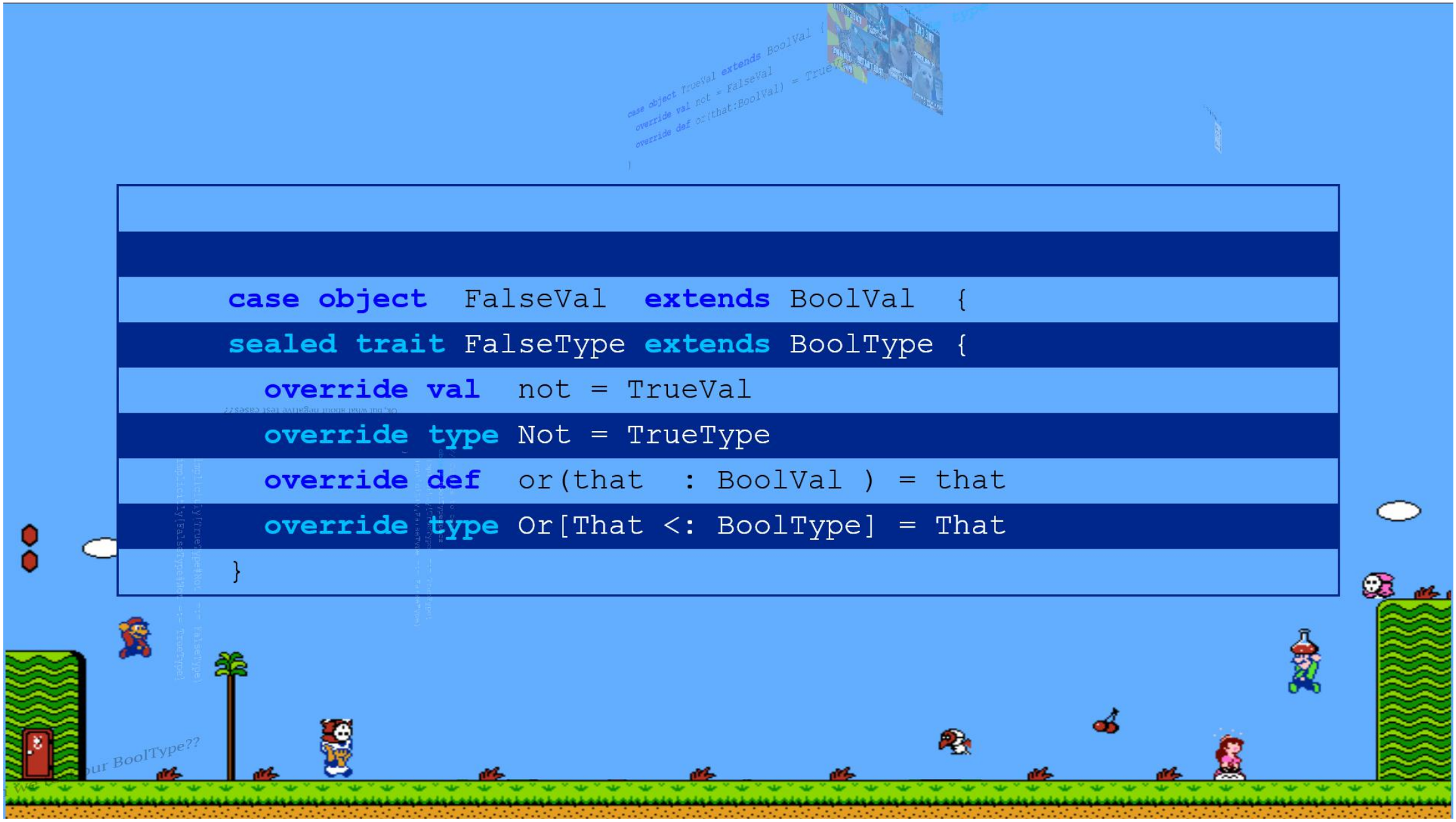

sealed trait BoolVal {
sealed trait BoolType {
def not : BoolVal
type Not <: BoolType
def or(that : BoolVal) : BoolVal
type Or[That <: BoolType] <: BoolType
}



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

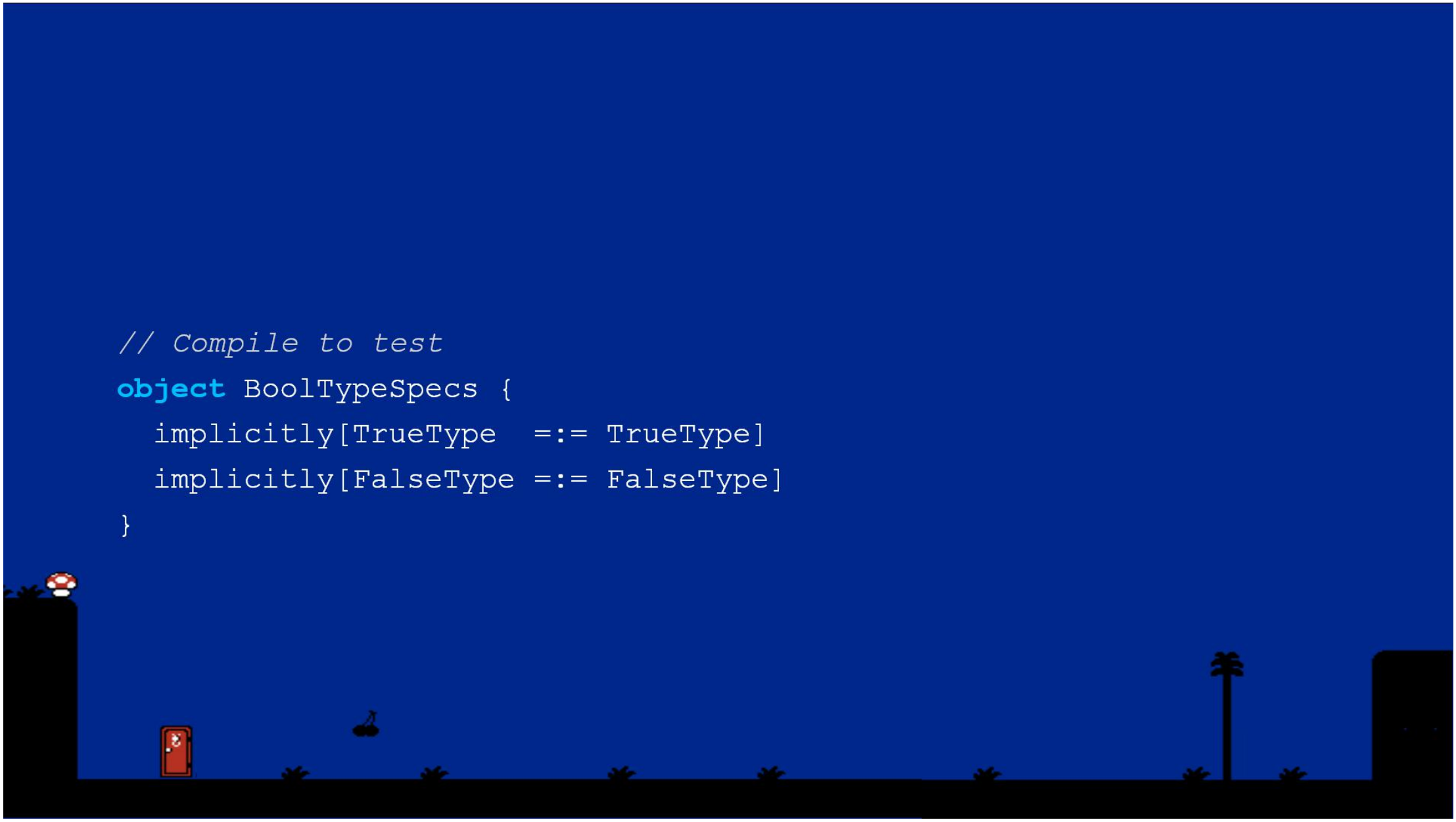

case object TrueVal extends BoolVal {
sealed trait TrueType extends BoolType {
override val not = FalseVal
override type Not = FalseType
override def or(that : BoolVal) = TrueVal
override type Or[That <: BoolType] = TrueType
}


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]
```

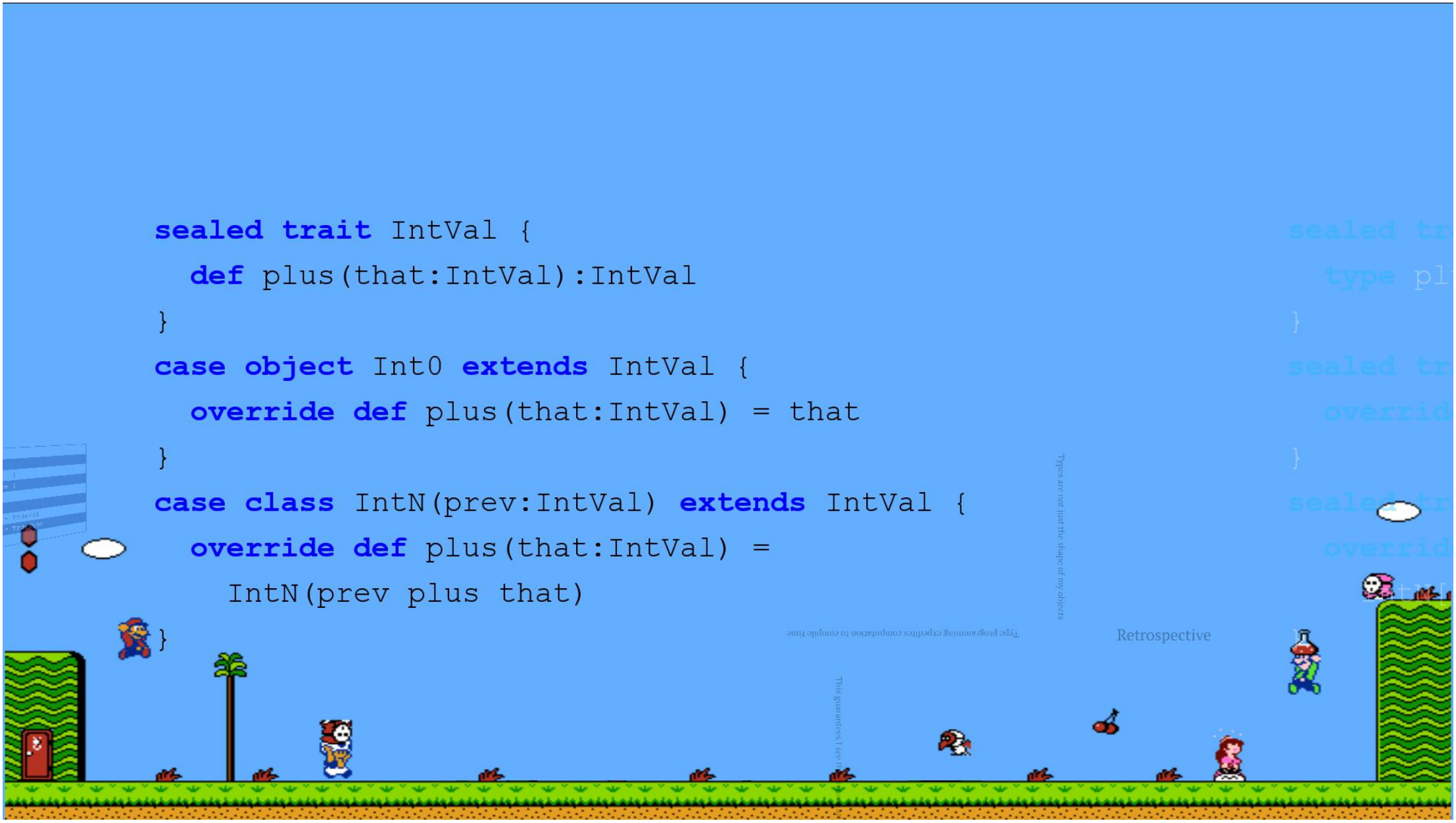
Ok, but what about negative test cases??


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

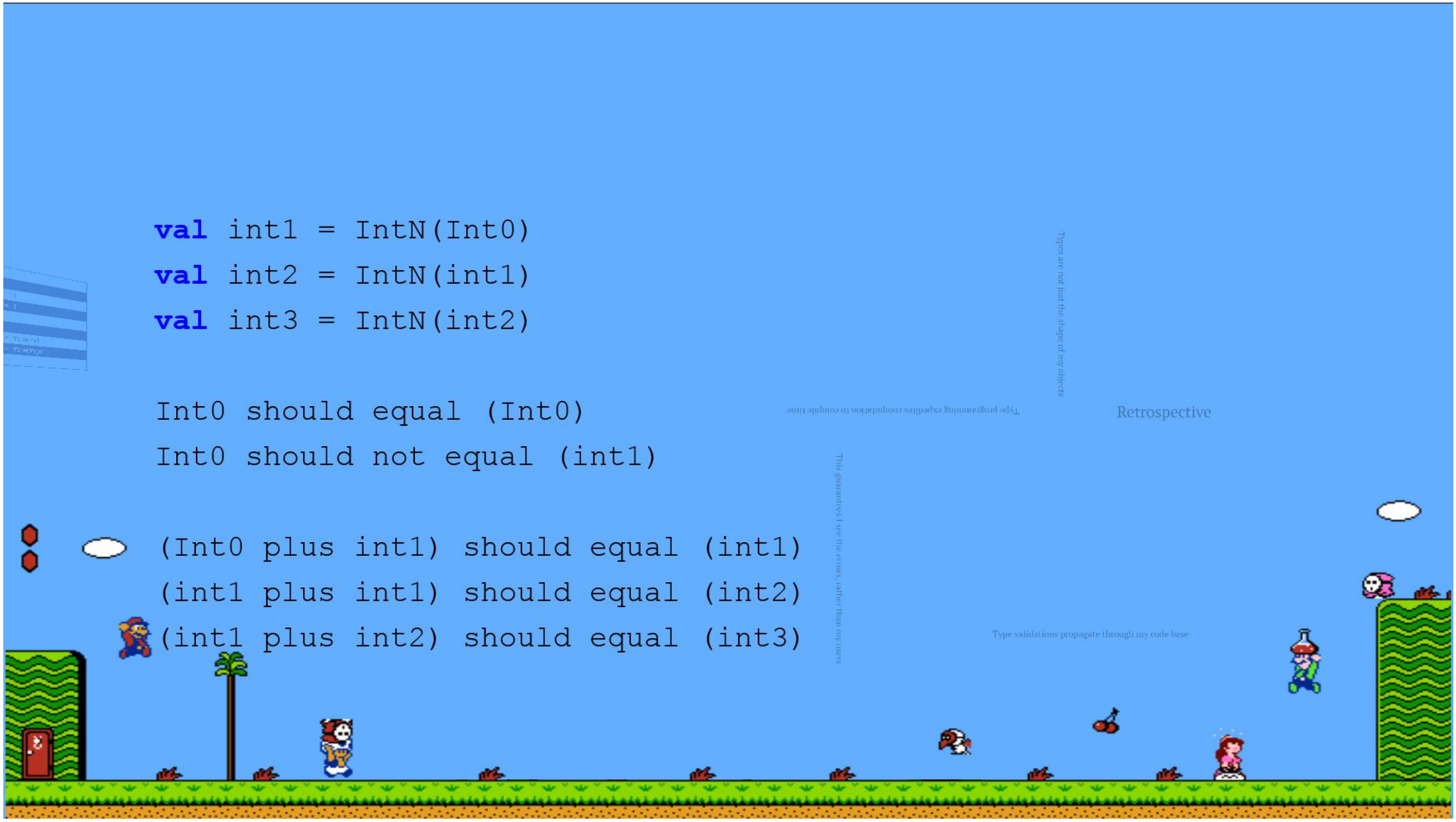

illTyped("implicitly[TrueType#Not =:= TrueType]")
illTyped("implicitly[FalseType#Not =:= FalseType]")
illTyped("implicitly[TrueType#Or[TrueType] =:= TrueType]")
illTyped("implicitly[TrueType#Or[FalseType] =:= TrueType]")

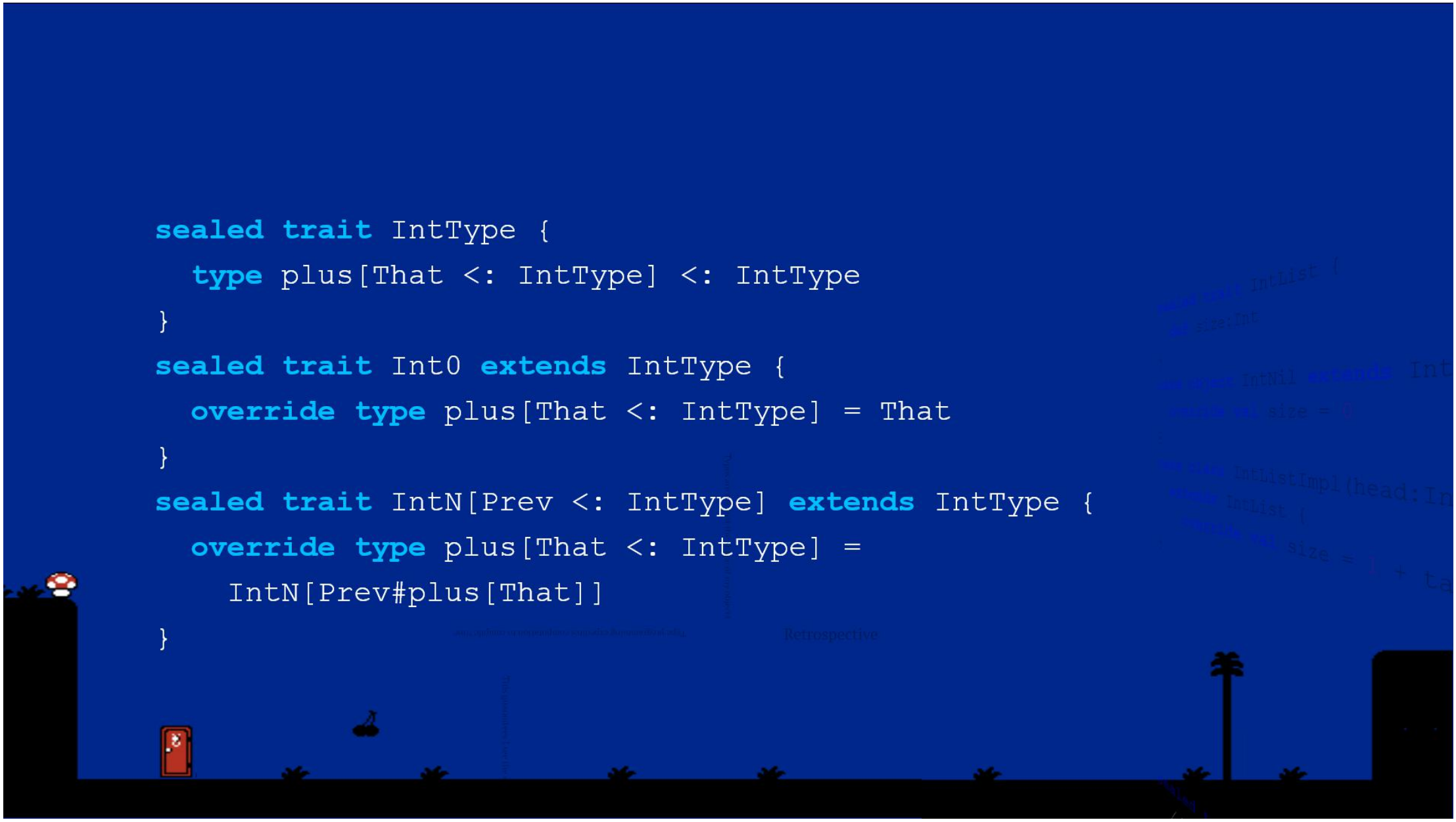

MARIO MADNESS
Nintendo
EUROPA-VERSION
type Int1 =
type Int2 =
type Int3 =
Well that was easy... It's just two types, really...
implicitly[
illTyped("i
implicitly[
implicitly[
implicitly[

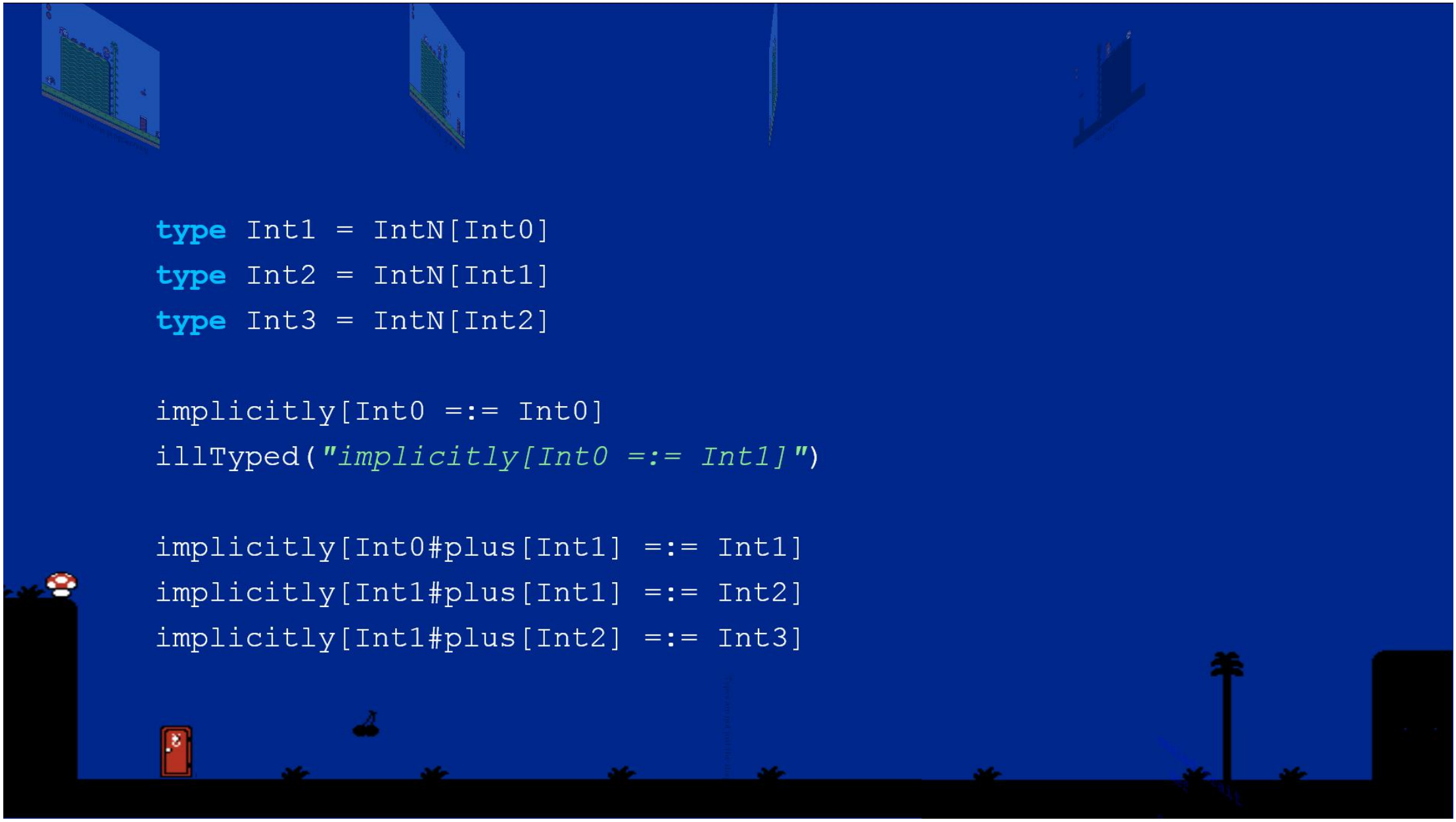
```
sealed trait IntVal {
  def plus(that:IntVal):IntVal
}
case object Int0 extends IntVal {
  override def plus(that:IntVal) = that
}
case class IntN(prev:IntVal) extends IntVal {
  override def plus(that:IntVal) =
    IntN(prev plus that)
}
```



```
val int1 = IntN(Int0)
val int2 = IntN(int1)
val int3 = IntN(int2)

Int0 should equal (Int0)
Int0 should not equal (int1)

(Int0 plus int1) should equal (int1)
(int1 plus int1) should equal (int2)
(int1 plus int2) should equal (int3)
```

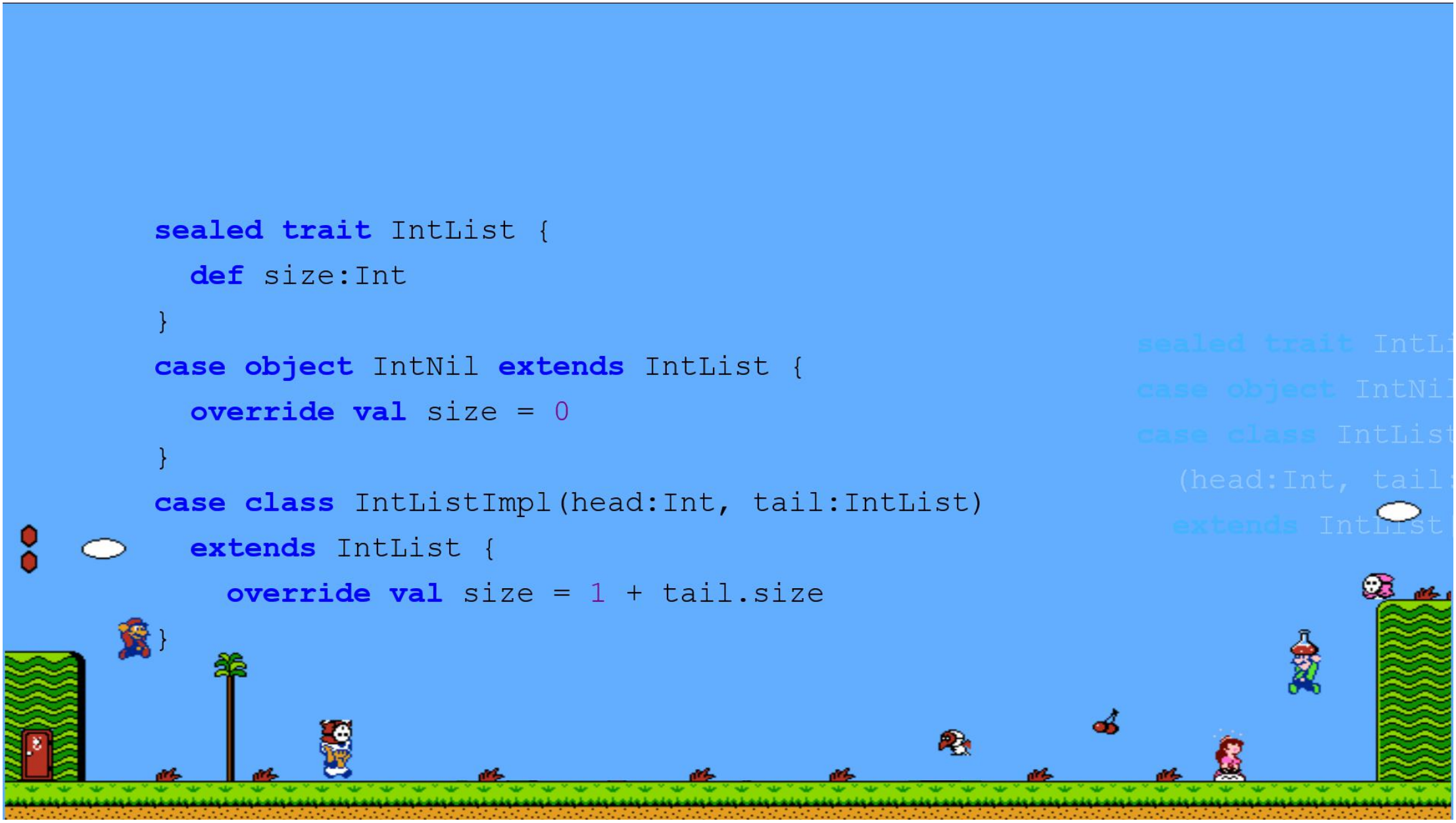



sealed trait IntType {
type plus[That <: IntType] <: IntType
}
sealed trait Int0 extends IntType {
override type plus[That <: IntType] = That
}
sealed trait IntN[Prev <: IntType] extends IntType {
override type plus[That <: IntType] =
IntN[Prev#plus[That]]
}

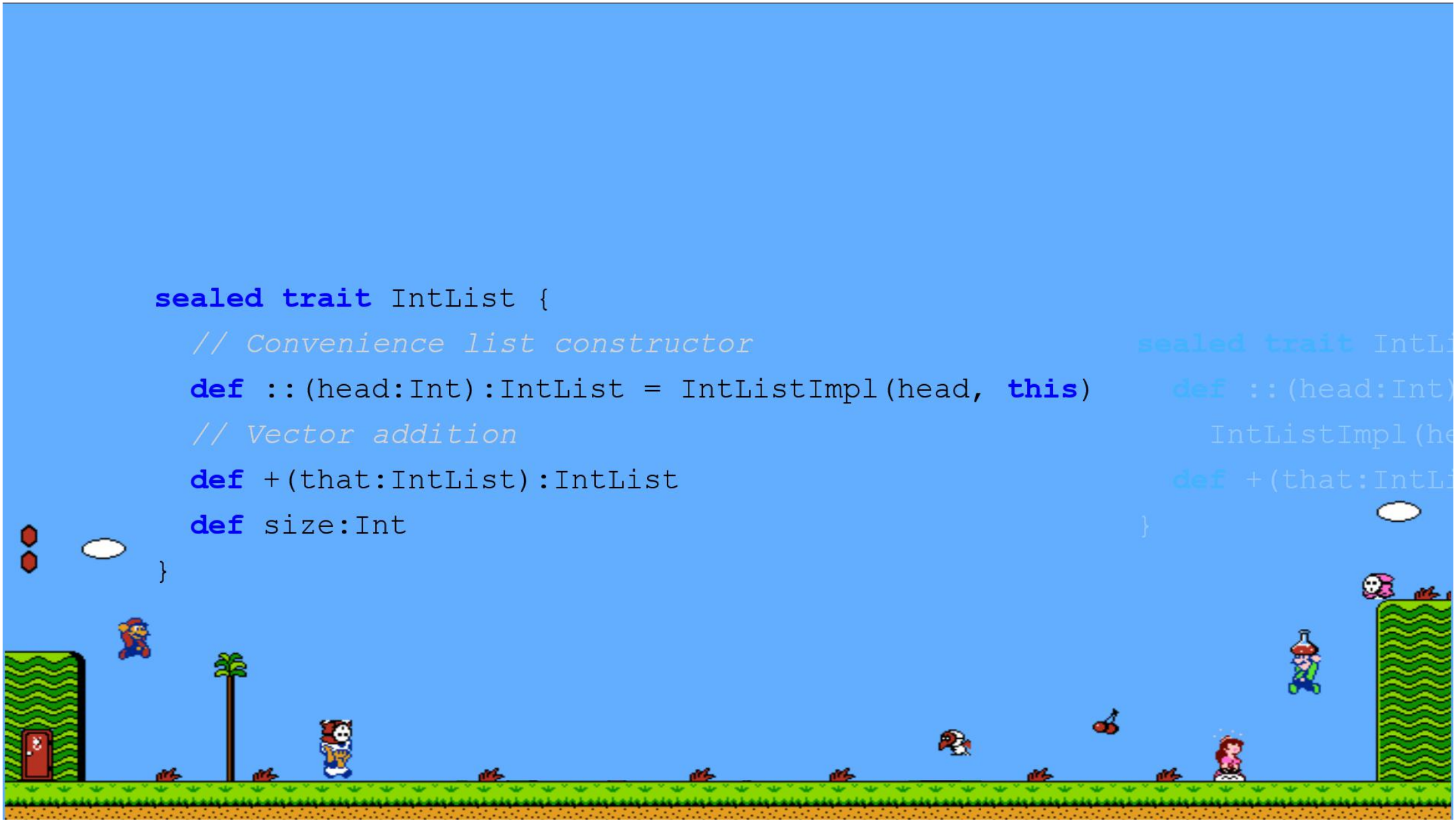

type Int1 = IntN[Int0]
type Int2 = IntN[Int1]
type Int3 = IntN[Int2]
implicitly[Int0 =:= Int0]
illTyped("implicitly[Int0 =:= Int1]")
implicitly[Int0#plus[Int1] =:= Int1]
implicitly[Int1#plus[Int1] =:= Int2]
implicitly[Int1#plus[Int2] =:= Int3]


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

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

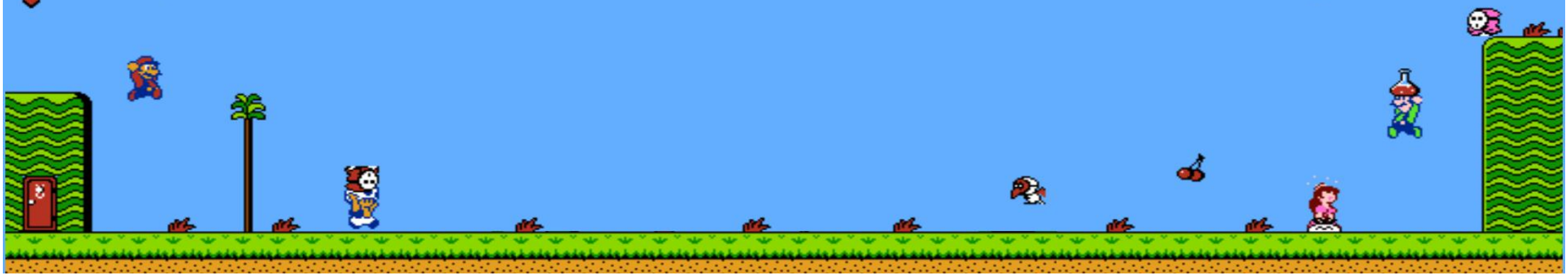


```
sealed trait IntList {
  // Convenience list constructor
  def ::(head:Int):IntList = IntListImpl(head, this)
  // Vector addition
  def +(that:IntList):IntList
  def size:Int
}
```

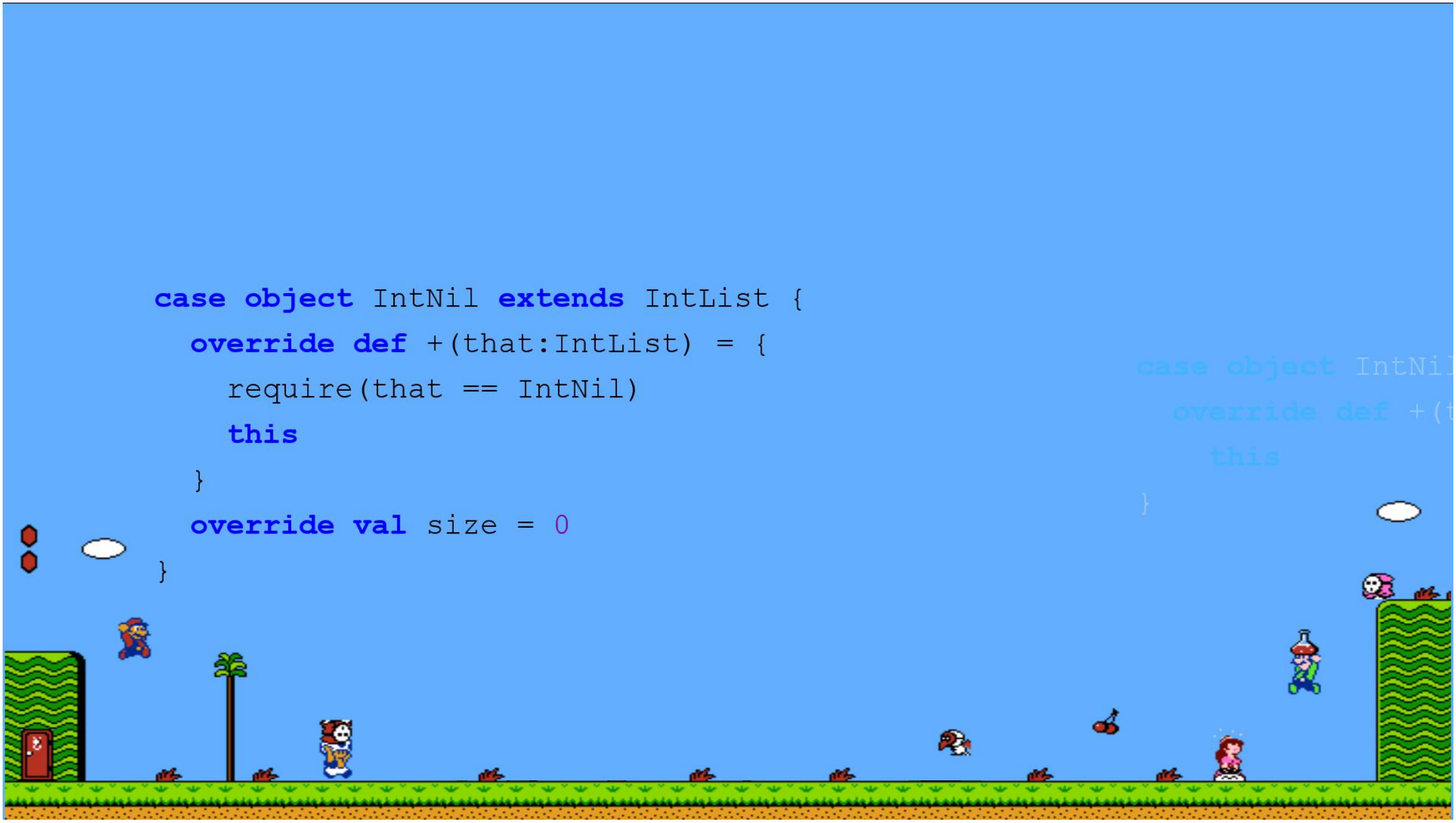


Type-Level Programming | The Subspace of Scala | by Joe Barnes @joescii - Mozilla Firefox

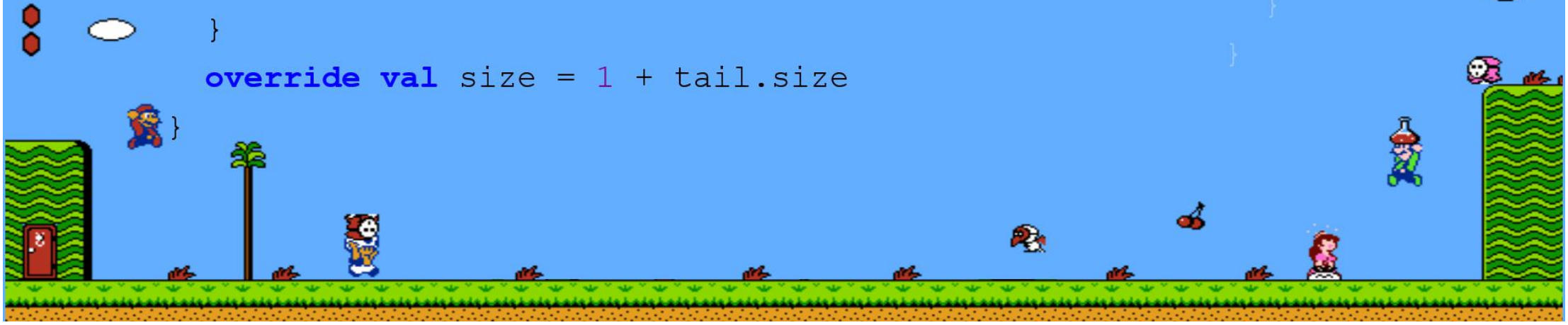
```
val sum = (1 :: 2 :: IntNil) + (3 :: 4 :: IntNil)
sum should equal (4 :: 6 :: IntNil)

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



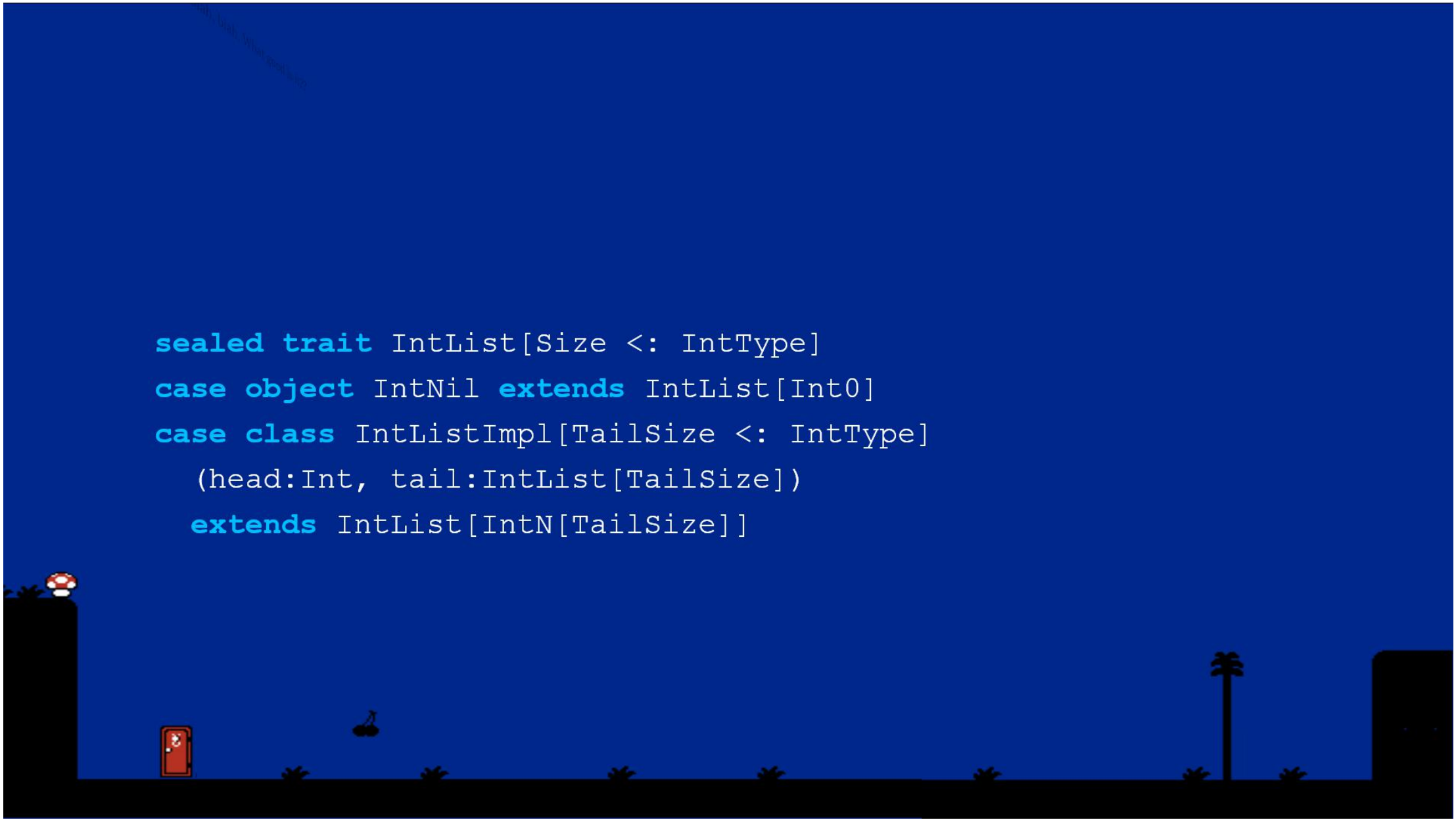
```
case object IntNil extends IntList {
  override def +(that:IntList) = {
    require(that == IntNil)
    this
  }
  override val size = 0
}
```



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



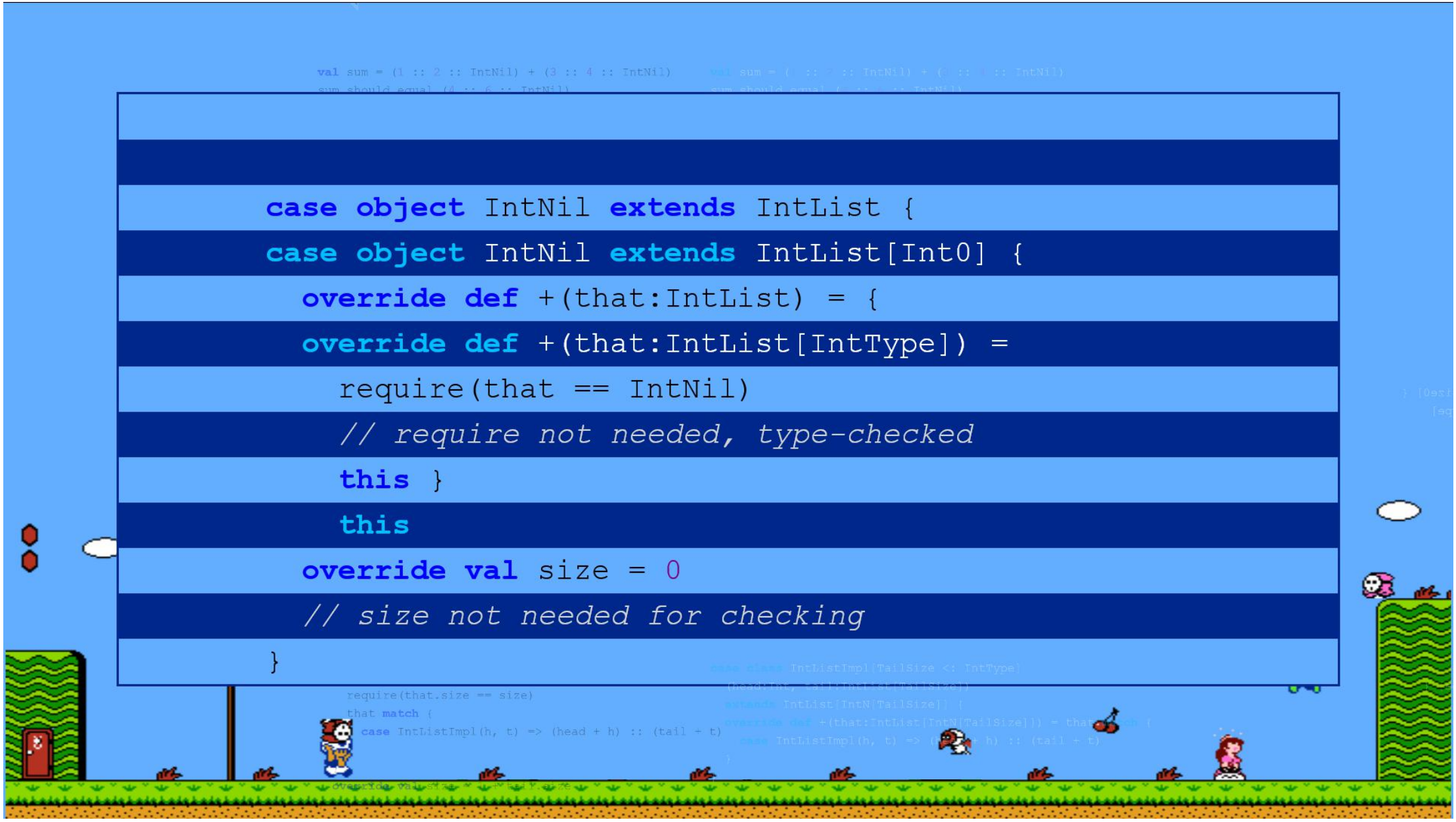
Compile time! (The lists are immutable)


sealed trait IntList[Size <: IntType]
case object IntNil extends IntList[Int0]
case class IntListImpl[TailSize <: IntType]
(head:Int, tail:IntList[TailSize])
extends IntList[IntN[TailSize]]


sealed trait IntList[Size <: IntType] {
this) def ::(head:Int):IntList[IntN[Size]] =
IntListImpl(head, this)
def +(that:IntList[Size]):IntList[Size]
}

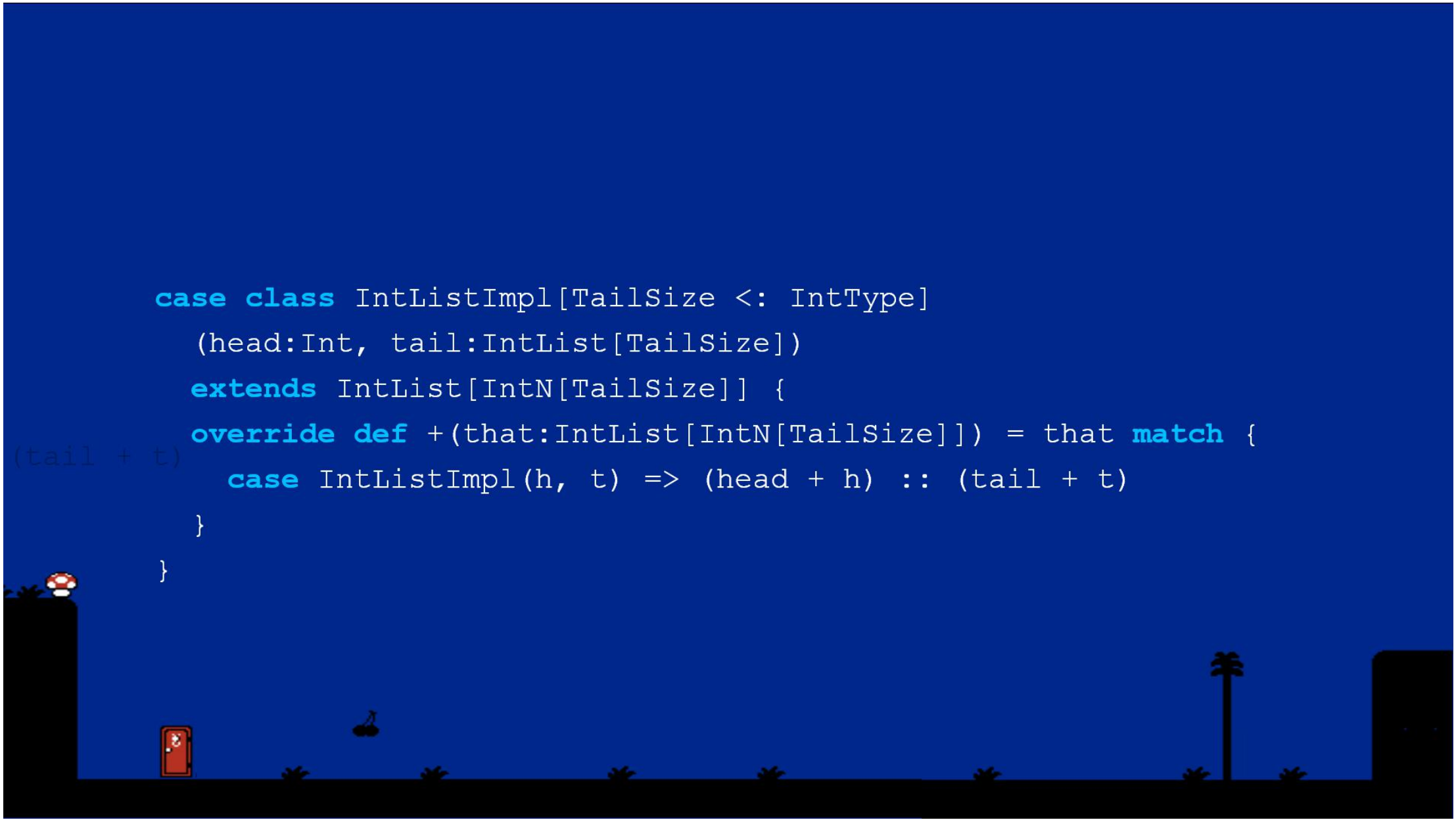

il)
val sum = (1 :: 2 :: IntNil) + (3 :: 4 :: IntNil)
sum should equal (4 :: 6 :: IntNil)
// Screw IllegalArgument Exception.
// This crap won't even compile!
illTyped("(1 :: 2 :: IntNil) + (5 :: IntNil)")

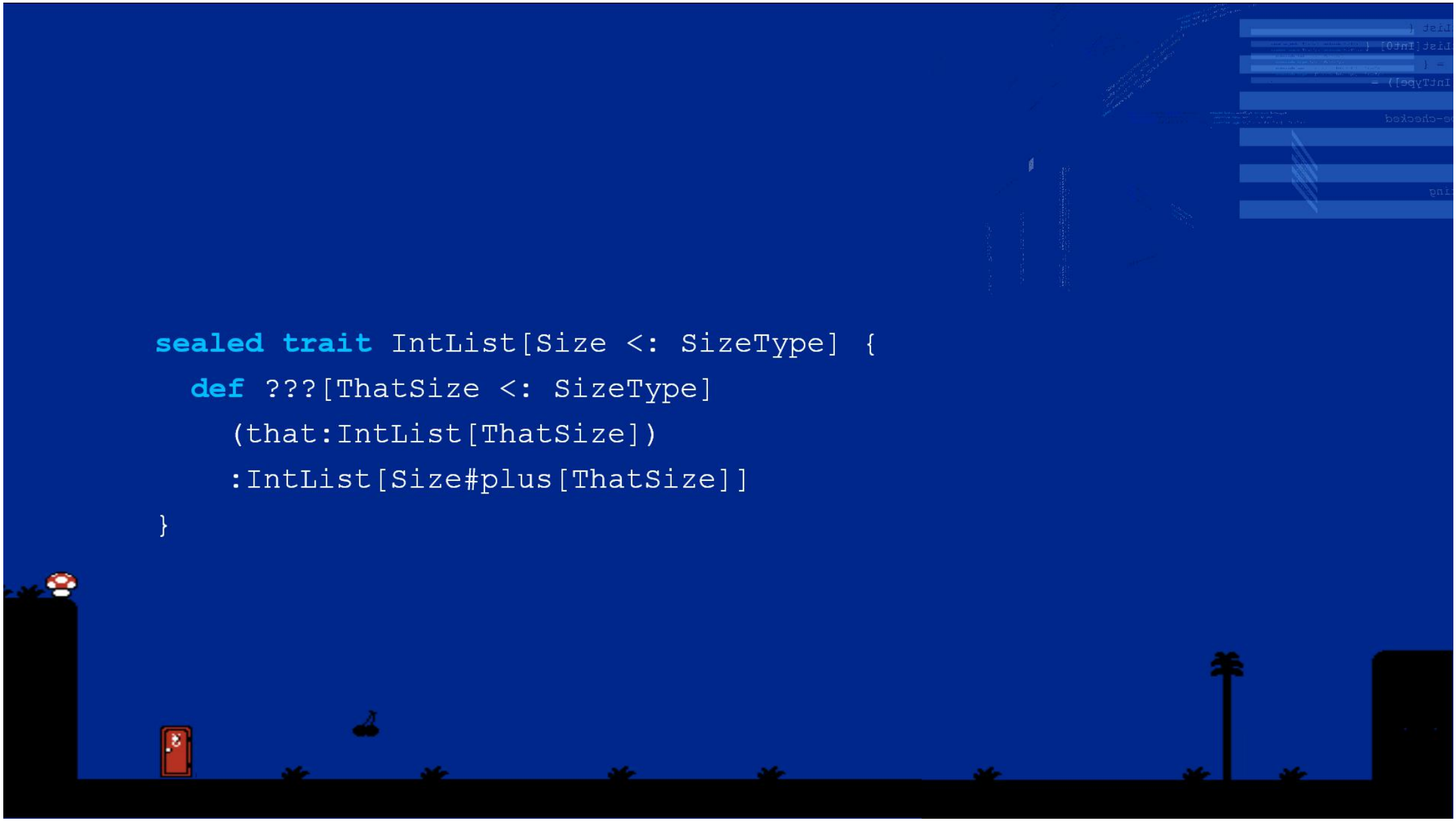
```
case object IntNil extends IntList[Int0] {
  override def +(that:IntList[IntType]) =
    this
}
```

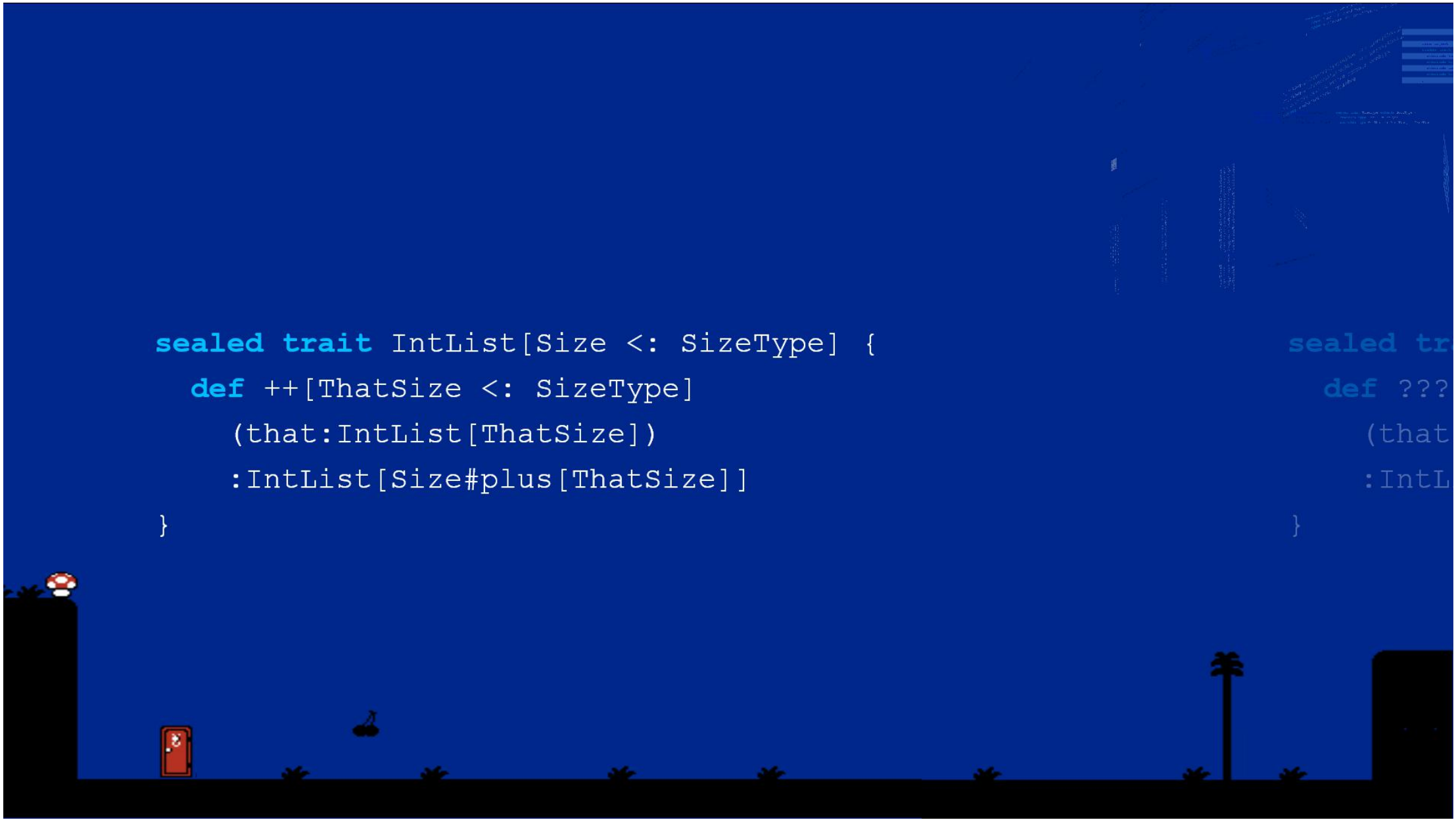

case object IntNil extends IntList {
case object IntNil extends IntList[Int0] {
override def +(that:IntList) = {
override def +(that:IntList[IntType]) =
require(that == IntNil)
// require not needed, type-checked
this }
this
override val size = 0
// size not needed for checking
}

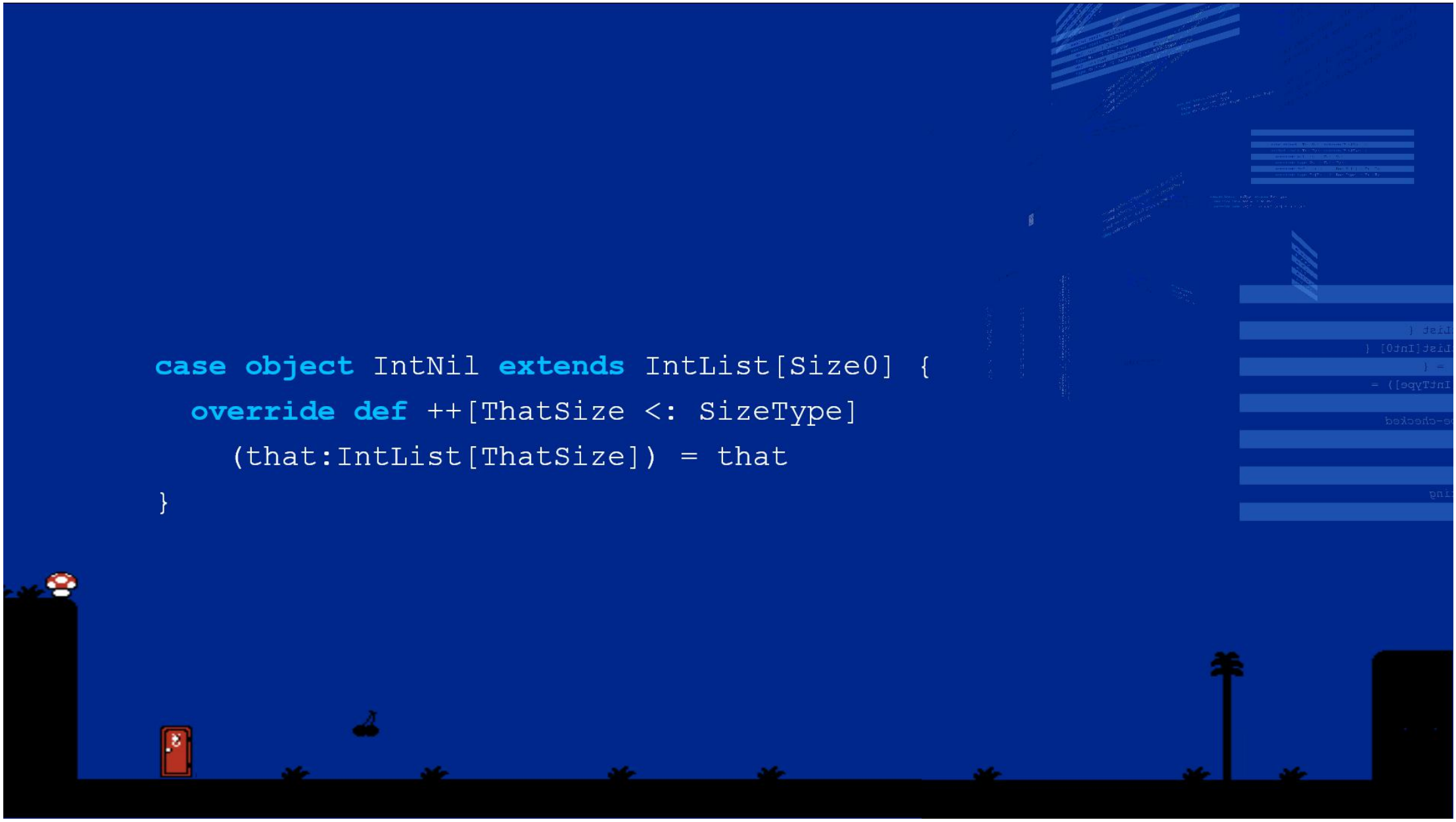
```
case class IntListImpl[TailSize <: IntType]
  (head:Int, tail:IntList[TailSize])
  extends IntList[IntN[TailSize]] {
  override def +(that:IntList[IntN[TailSize]]) = that match {
    case IntListImpl(h, t) => (head + h) :: (tail + t)
  }
}
```

(tail + t)




sealed trait IntList[Size <: SizeType] {
def ???[ThatSize <: SizeType]
(that:IntList[ThatSize])
:IntList[Size#plus[ThatSize]]
}


sealed trait IntList[Size <: SizeType] {
def ++[ThatSize <: SizeType]
(that:IntList[ThatSize])
:IntList[Size#plus[ThatSize]]
}
sealed tr
def ???
(that
:IntL
}


case object IntNil extends IntList[Size0] {
override def ++[ThatSize <: SizeType]
(that:IntList[ThatSize]) = that
}


case class IntListImpl[TailSize <: IntType]
(head:Int, tail:IntList[TailSize])
extends IntList[IntN[TailSize]] {
override def ++[ThatSize <: SizeType]
(that:IntList[ThatSize]) =
IntListImpl(head, tail++that)
}


val sum =
((1 :: 2 :: IntNil) ++ (3 :: IntNil)) +
(4 :: 5 :: 6 :: IntNil)
sum should equal (5 :: 7 :: 9 :: IntNil)

Retrospective

Retrospec

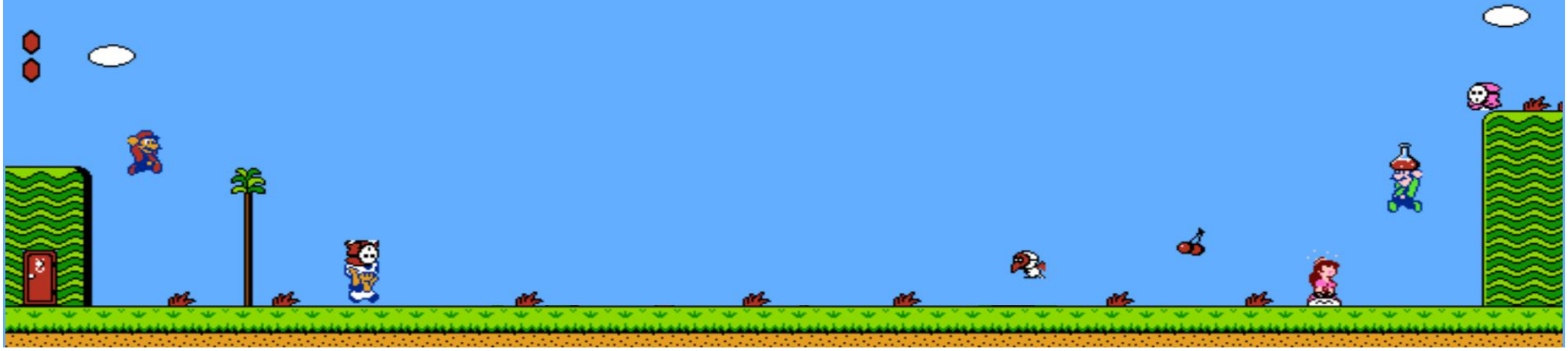
Type programmin

Types are not just the shape of my objects

This guarantees I se

Type programming expedites computation to compile time

This guarantees I see the errors, rather than my users



Type validations propagate through my code base

Types are not just the shape of my objects

Type programming expedites computation to compile time

Retrospective

This guarantees the errors, rather than my users

Questions?

Type validations propagate through my code base


erns?
Type programming expedites computation to compile time
Questions?
Types are not just the shape of my objects
Retrospective
This guarantees I see the error sooner than my users
Comments?
Type validations propagate through my code base
Download slides
Presentation Source

Types are not just the shape of my objects

Type programming expedites computation to compile time

Retrospective

This guarantees I see the errors, rather than my users

Concerns?

Questions?

Type validations propagate through my code base

Comments?

Questions?

Comments?

Retrospective

Type validations propagate through my code base

Download slides

Presentation Source