

# 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??

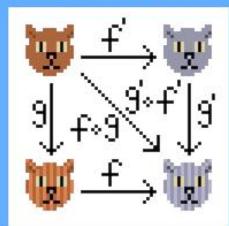
Not much...

So... Type programming...  
What do you know about *that*?

I've made zero contributions to shapeless

Not much...

Also no contributions to Cats



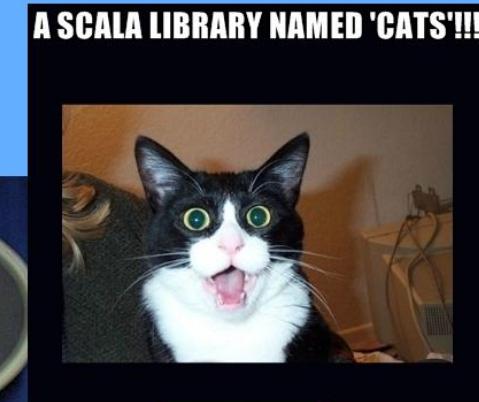
## Type-Level Programming

*The Subspace of Scala*

Joe Barnes

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

Although I have contributed plenty of cat memes!

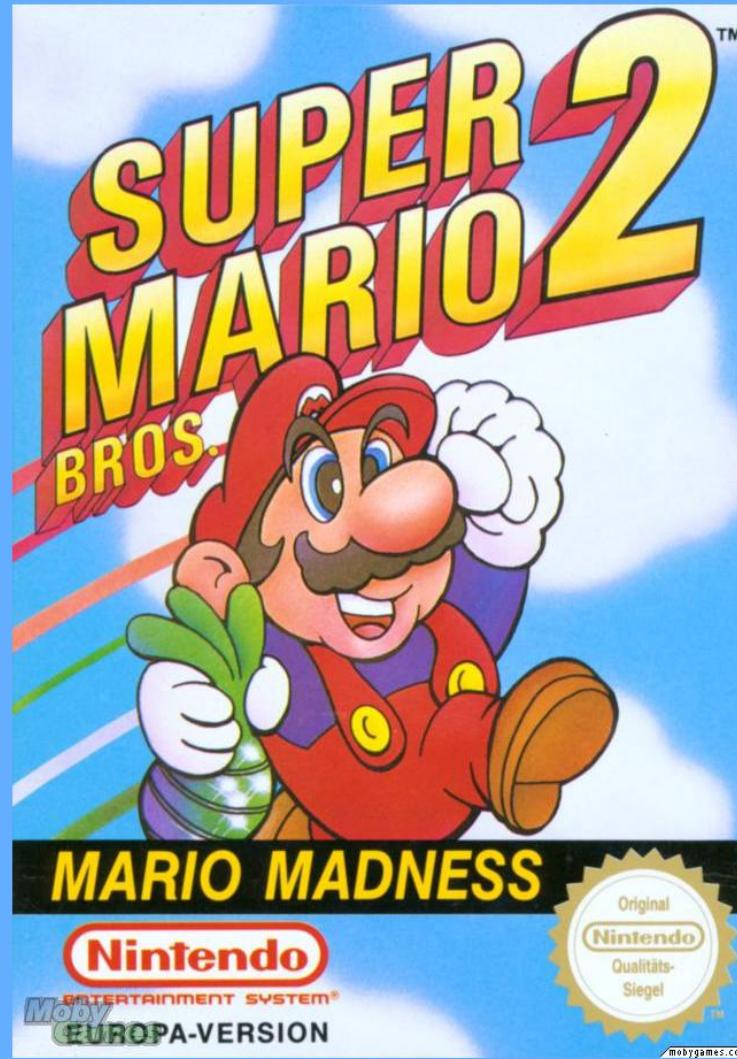


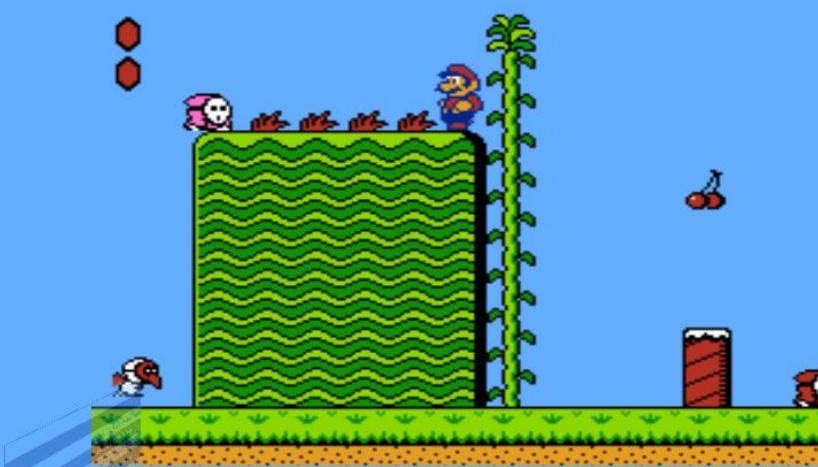
primarily Java from 2004-2013  
started Scala late 2012  
started Scala at Mentor Graphics late 2013  
committer in July 2014

sealed t  
case obj  
case obj

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

Programming in Scala is like...





Normal value programming



A flask of type programming!



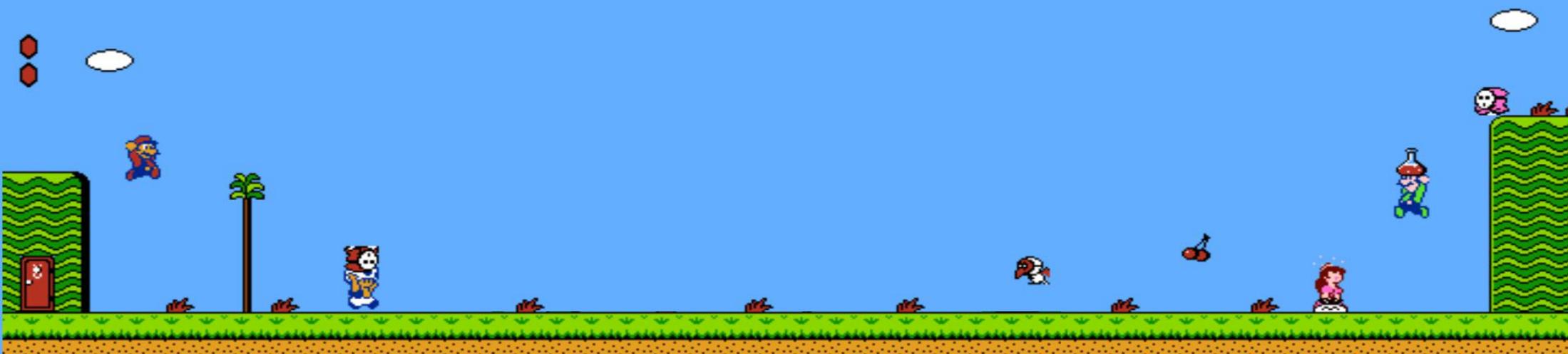
Where does this go?

...the first floor  
of the castle  
is the last floor  
of the castle



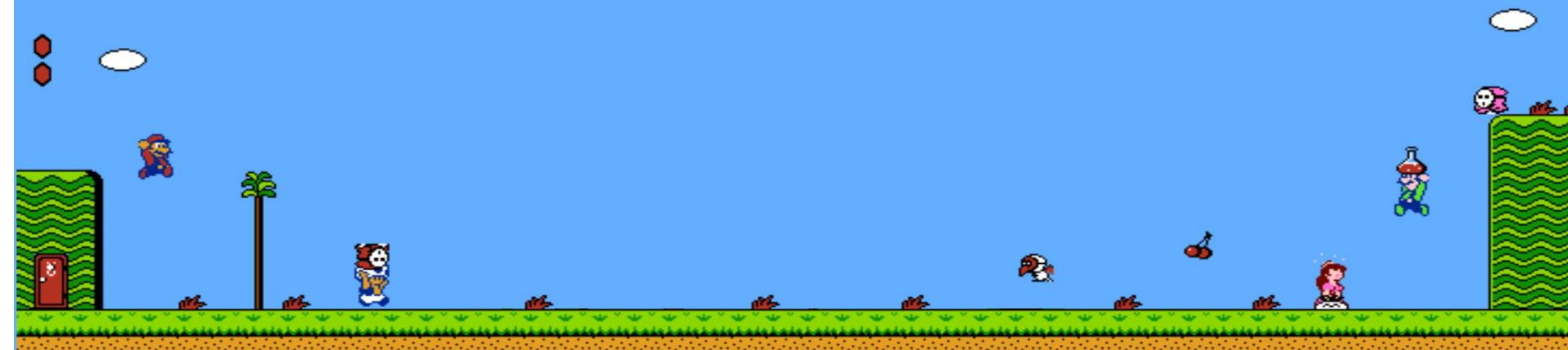
SUBSPACE!

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

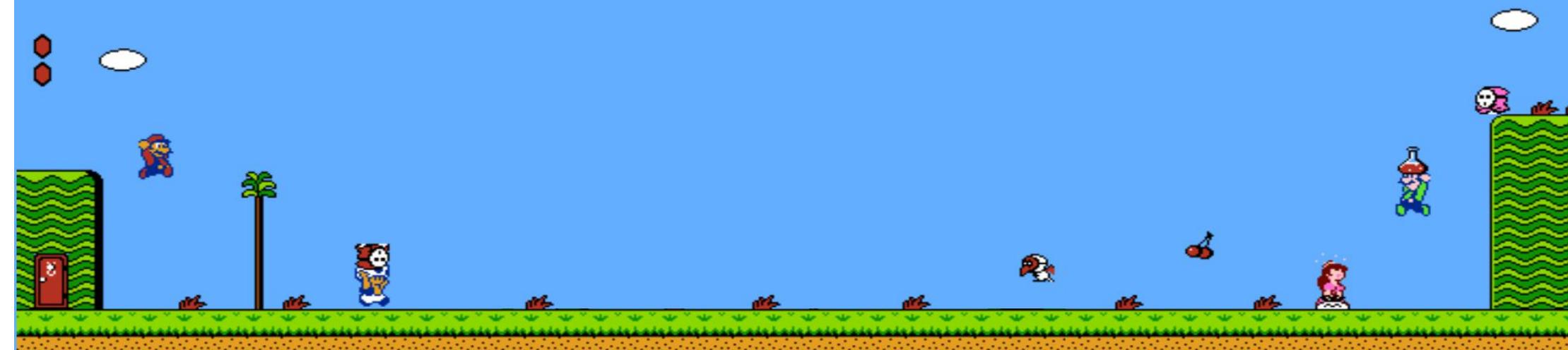


```
val num = 1 + 2 + 3
```

```
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]
```

The background of the image is a screenshot from the video game Super Mario Bros. It shows a dark blue sky with a single white star. Below is a black ground with several green plants and small brown mushrooms. A red door is visible on the left, and a palm tree stands on the right. The overall aesthetic is that of a classic 8-bit video game.

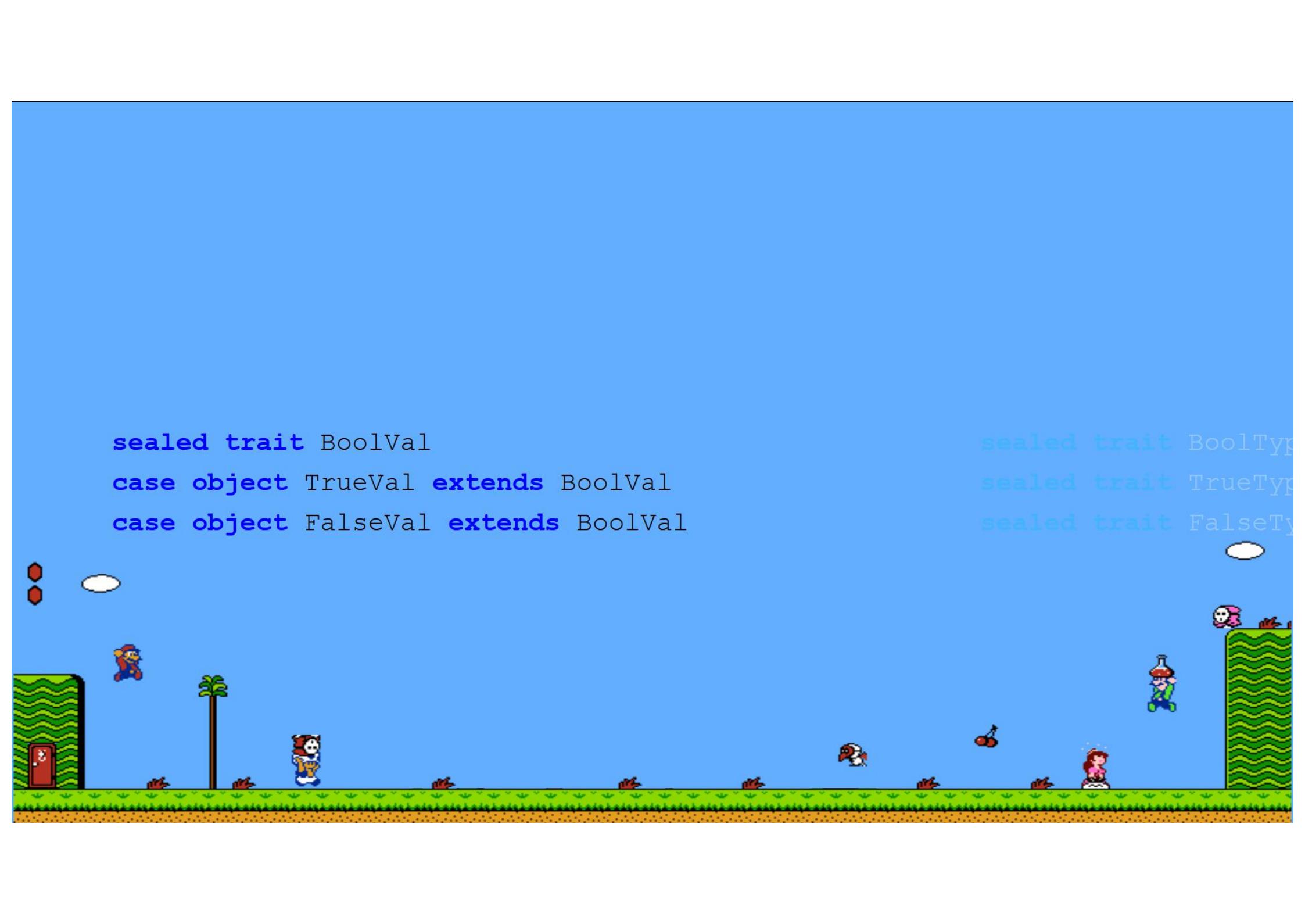
```
val now = new Date
```

```
Lazy val str = "goooooood"
```

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

Programming in Scala is like...





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

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



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

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

case object Fas  
case object Fa

Sealed trait  
BoolType

BoolVal

def not:BoolVal

def or(that:BoolVal):BoolVal

case object Fas  
case object Fa

case object Fas  
case object Fa

BoolVal

BoolType

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  
}
```

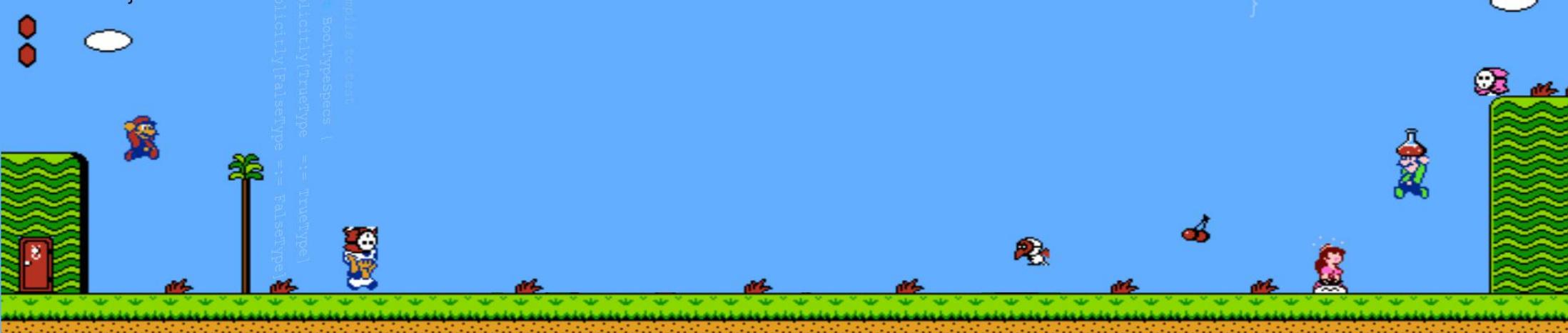
```
sealed trait TrueType extends BooleanType
  override type Not = FalseType
  override type Or[That <: BooleanType] = That
}
```

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

```
}
```

```
// Compiler to test  
// BooleanTypeSpecs  
implicit(BooleanType := BooleanType)  
implicit(FalseType := FalseType)
```

```
sealed trait Fa  
override type  
override type  
}
```



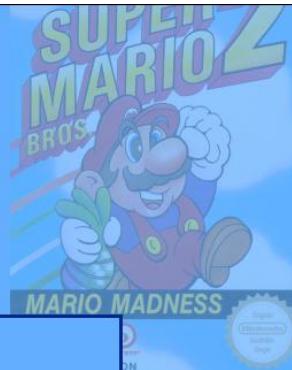
```
sealed trait BoolType  
sealed trait TrueType extends BoolType  
sealed trait FalseType extends BoolType
```



joe Barnes  
@joesciii  
prose :: 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  
not <: BoolType  
Or[That <: BoolType] <: BoolType

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

```
TrueType extends BoolType {  
    type FalseType extends BoolType {  
        type And[That <: BoolType] = TrueType  
        type Or[That <: BoolType] = FalseType  
    }  
}
```

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

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

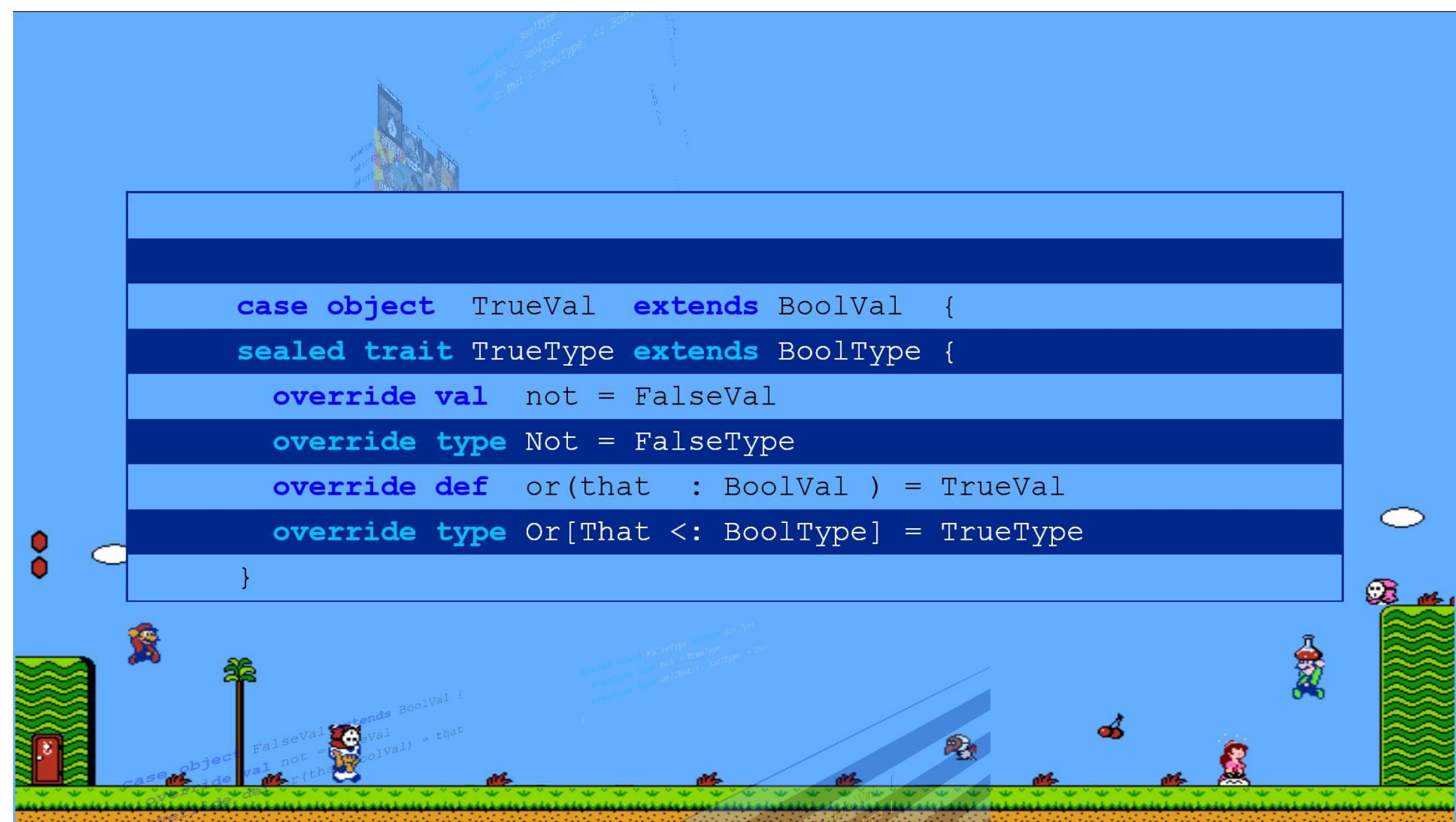
```
sealed trait BoolType {  
  type TrueType <: BoolType  
  type FalseType <: BoolType  
  type NotType <: BoolType
```



```
{  
  sealed trait TrueType extends BoolType {  
    override type Not = FalseType  
  }  
  trueVal 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  
}
```

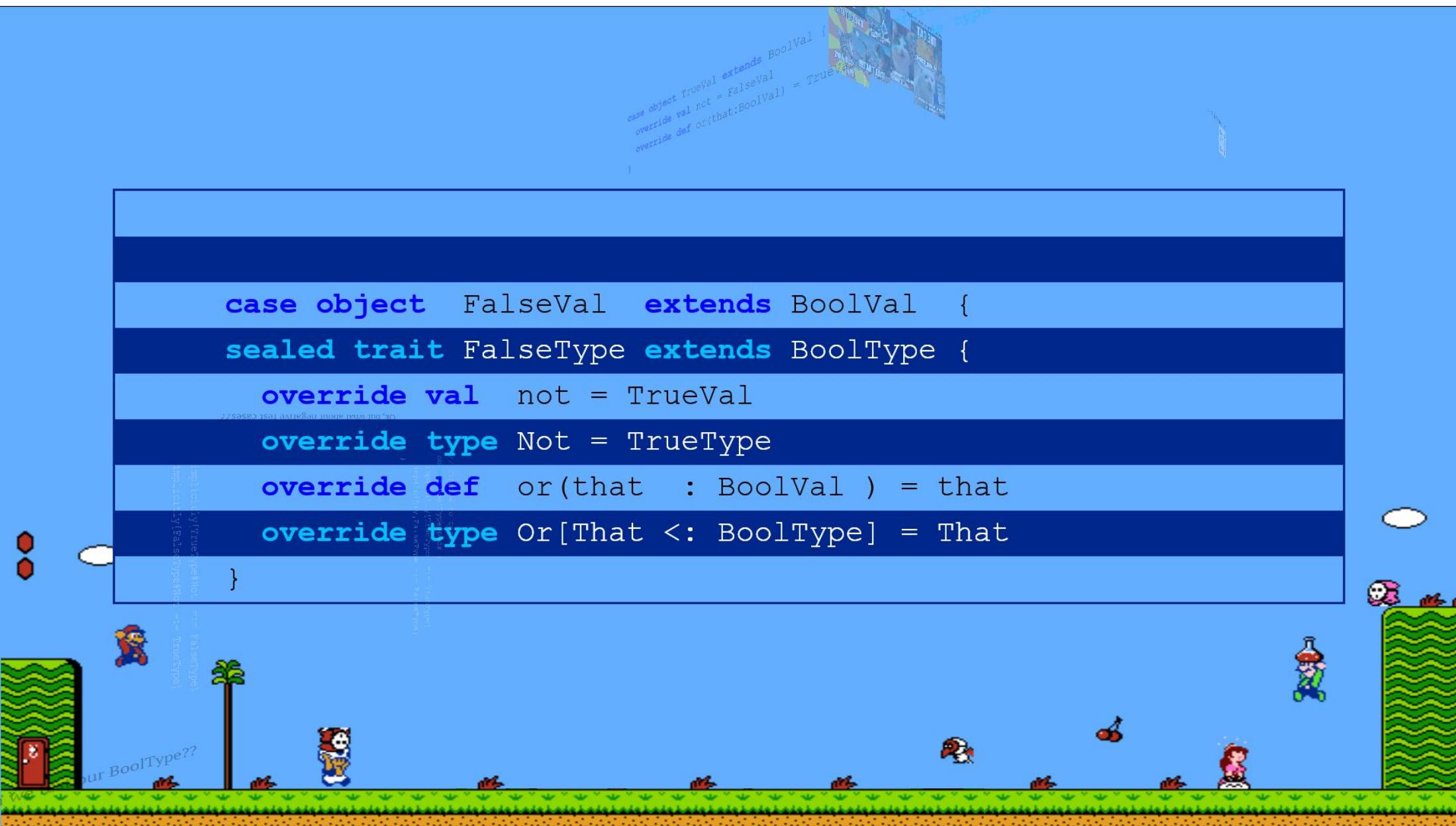


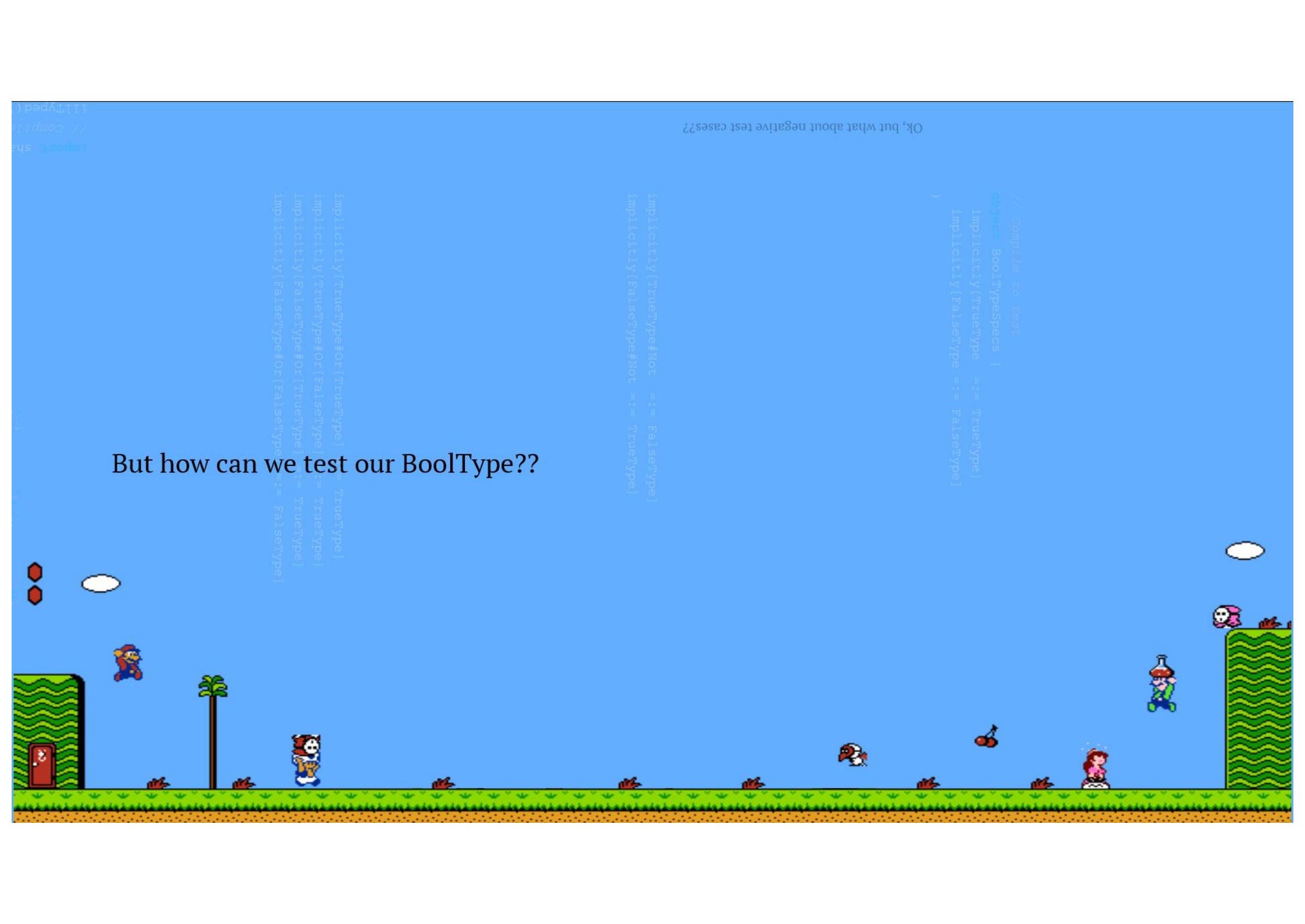
```
case object FalseVal extends BoolVal {  
    override val not = FalseVal  
    override def or(that:BoolVal) = TrueVal  
}  
case object TrueVal extends BoolVal {  
    override val not = TrueVal  
    override def or(that:BoolVal) = TrueVal  
}
```

```
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  
    }  
    }
```

Implicitly[true:TypeNot]  
Implicitly[FalseType] => FalseType

our BoolType??





But how can we test our BoolType??

Ok, but what about negative test cases??

```
// Compile to best
class BoolTypeSpec {
    implicitly[TrueType ==:= TrueType]
    implicitly[FalseType ==:= FalseType]
    implicitly[FalseType !=:= TrueType]
}
```

```
implicity[TrueType#not ==:= FalseType]
implicity[FalseType#not ==:= TrueType]
```

```
implicity[TrueType#or[TrueType] ==:= TrueType]
implicity[TrueType#or[FalseType] ==:= TrueType]
implicity[FalseType#or[TrueType] ==:= TrueType]
implicity[FalseType#or[FalseType] ==:= FalseType]
```

```
littleped()
// Compile to best
implicity[TrueType ==:= TrueType]
implicity[FalseType ==:= FalseType]
```

```
// 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]
```

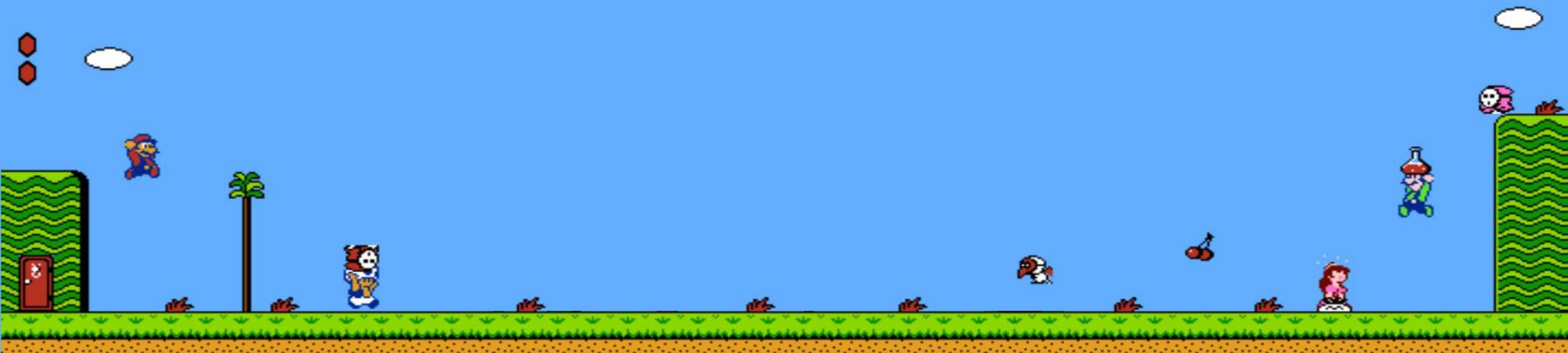


```
}
```

Ok, but what about negative test cases??

```
implicitly[True]  
implicitly[False]
```

```
implicitly[True]
```

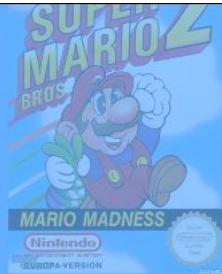




```
import shapeless.test.illTyped
// Compiles only if string DOESN'T compile
illTyped("implicitly[TrueType  ==:= FalseType]")
illTyped("implicitly[FalseType ==:= TrueType]")
```

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





```
type Int1 =  
type Int2 =  
type Int3 =
```

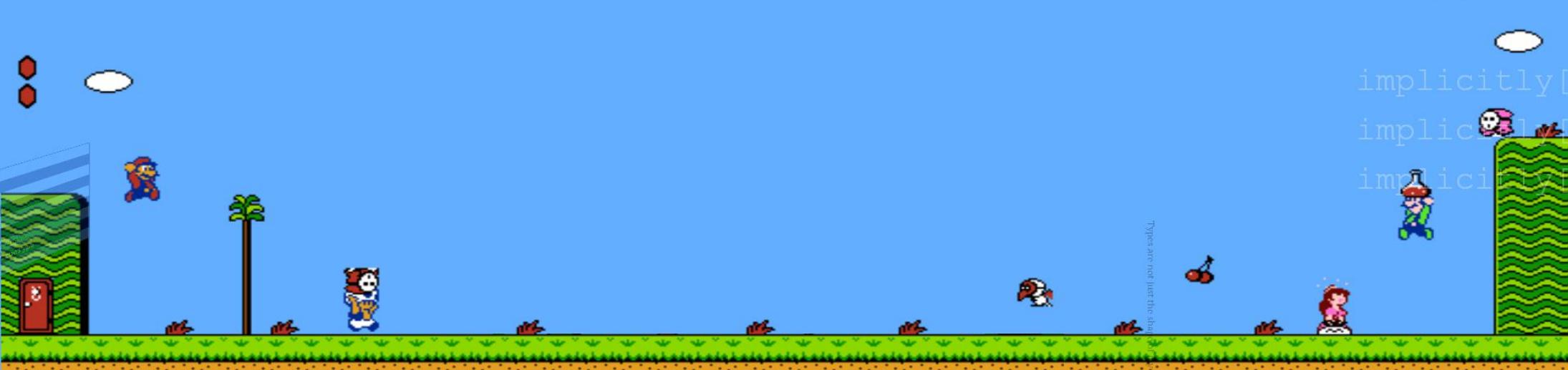
```
implicitly[  
illTyped("i")]
```

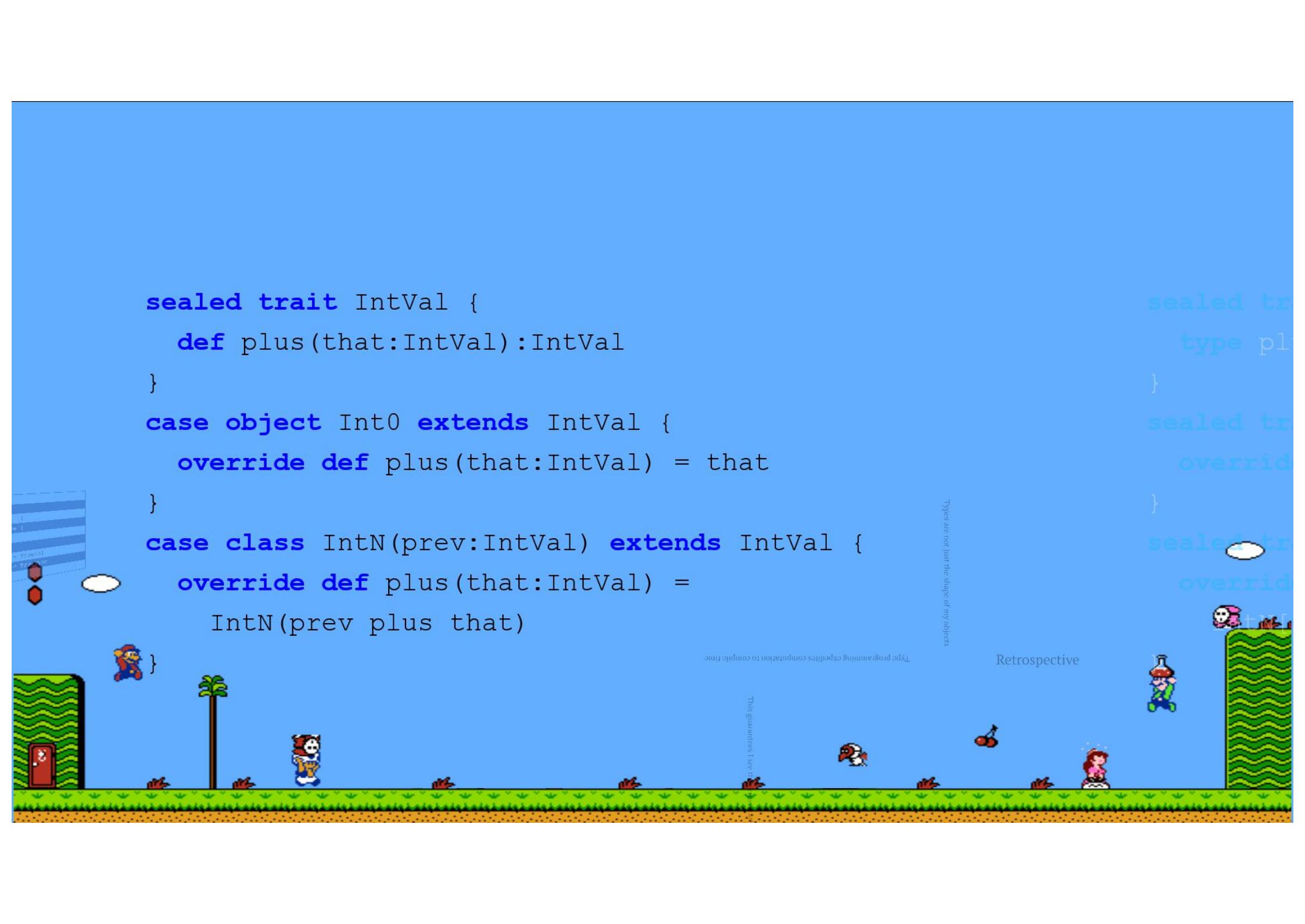
```
implicitly[  
implicitly[  
implicitly[  
implicitly[
```

```
implicitly[  
implicitly[  
implicitly[  
implicitly[
```

Well that was easy... It's just two types, really...

Type hints are not just the static





```
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)  
}
```

Type programming explores computation to complete the game

This guarantees I can

Types are not just the shape of my objects

Retrospective

```
sealed trait IntVal {  
    type plus(that:IntVal) = IntVal  
}  
  
sealed trait Int0 extends IntVal {  
    override def plus(that:IntVal) = that  
}  
  
sealed trait 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)

Type programming enables compilation to compile time

Retrospective

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

```
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]]  
}
```

www.claudiopizzo.com  
@claudiopizzo

Retrospective



```
type Int1 = IntN[Int0]
type Int2 = IntN[Int1]
type Int3 = IntN[Int2]
```

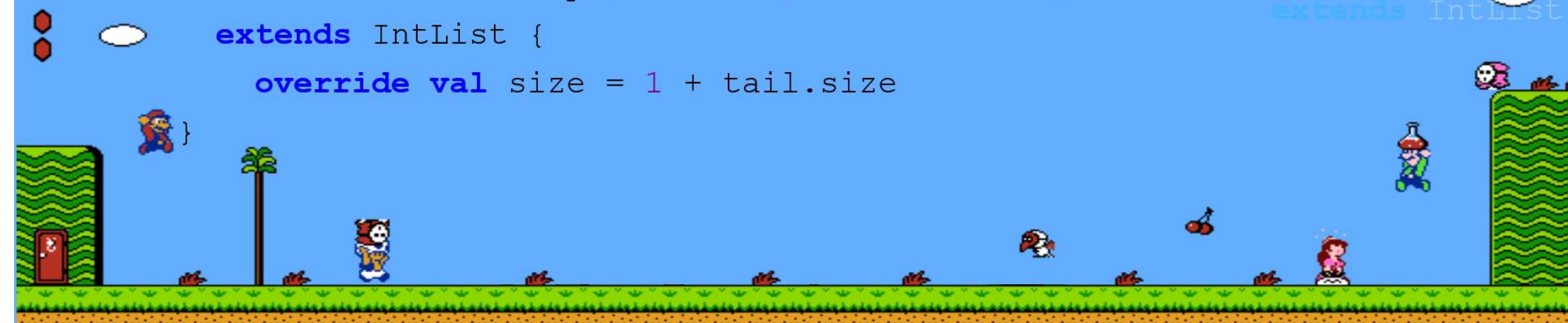
```
implicitly[Int0 ==:= Int0]
illTyped("implicitly[Int0 ==:= Int1]")
```

```
implicitly[Int0#plus[Int1] ==:= Int1]
implicitly[Int1#plus[Int1] ==:= Int2]
implicitly[Int1#plus[Int2] ==:= Int3]
```



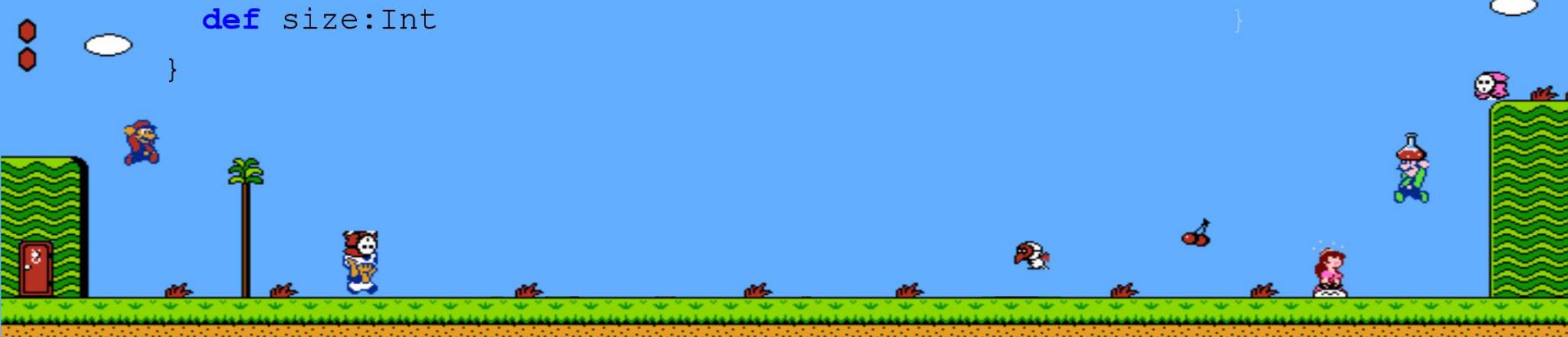
```
sealed trait IntList {  
    def size:Int  
}  
  
case object IntNil extends IntList {  
    override val size = 0  
}  
  
case class IntListImpl(head:Int, tail:IntList)  
extends IntList {  
    override val size = 1 + tail.size
```

```
sealed trait IntList  
case object IntNil extends IntList  
case class IntListImpl(head:Int, tail:IntList)  
    extends IntList
```



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

```
sealed trait IntList {  
    def ::(head:Int):IntList = IntListImpl(head, this)  
    def +(that:IntList):IntList  
}
```



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

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

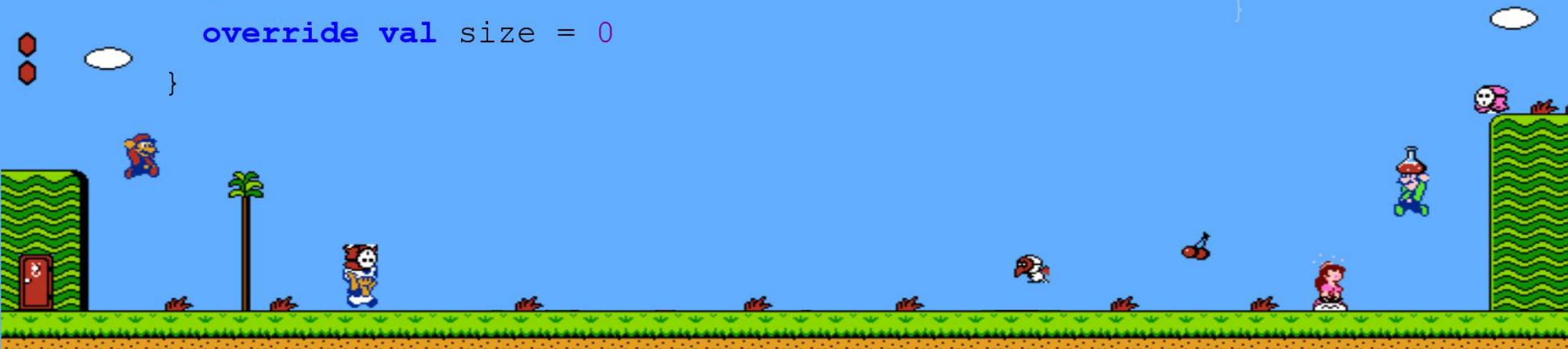
```
val sum = (1 :: 2  
sum should equal
```

```
// Screw IllegalArgumentException  
// This crap won't  
illTyped("((1 :: 2  
))")
```



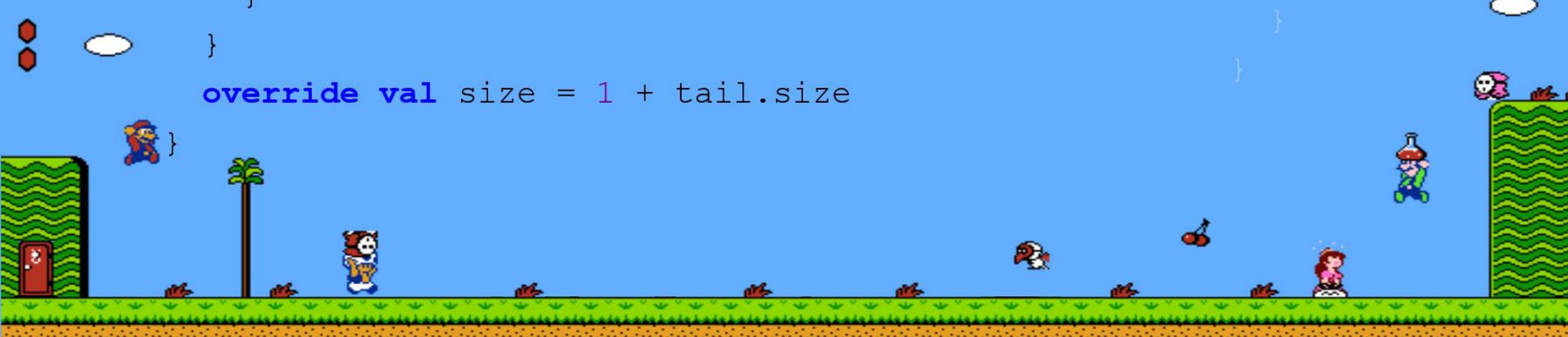
```
case object IntNil extends IntList {  
    override def +(that:IntList) = {  
        require(that == IntNil)  
        this  
    }  
    override val size = 0  
}
```

```
case object IntNil  
override def +(t  
this  
}  
}
```



```
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  
  }
```

```
case class IntList  
(head:Int, tail:  
  extends IntList  
  override def +(t:  
    case IntListIn  
    }  
    }  
  )  
  }
```



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]): IntList[IntType] =  
        this  
}
```



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

```
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  
  
  }  
  
  require(that.size == size)  
  that match {  
    case IntListImpl(h, t) => (head + h) :: (tail + t)  
  }  
}
```

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

) [0sec]  
[sq]



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



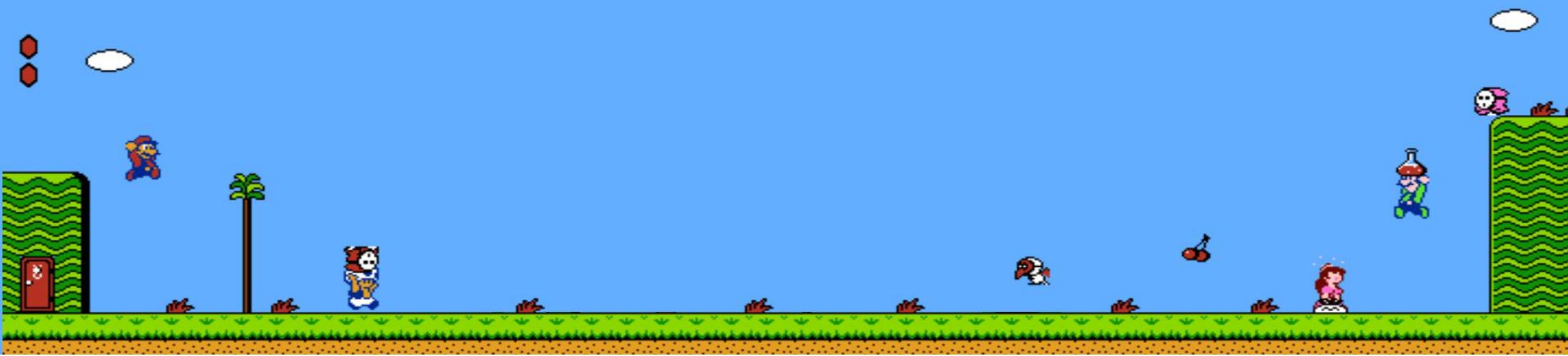
```
case class IntListImpl[TailSize <: IntType]  
  (head:Int, tail:IntList[TailSize])  
  extends IntList[IntN[TailSize]] {  
  
  override def ++[ThatSize <: SizeType]  
  (that:IntList[ThatSize]) =  
    IntListImpl(head, tail++that)  
}
```

case obj  
overrid  
(that

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



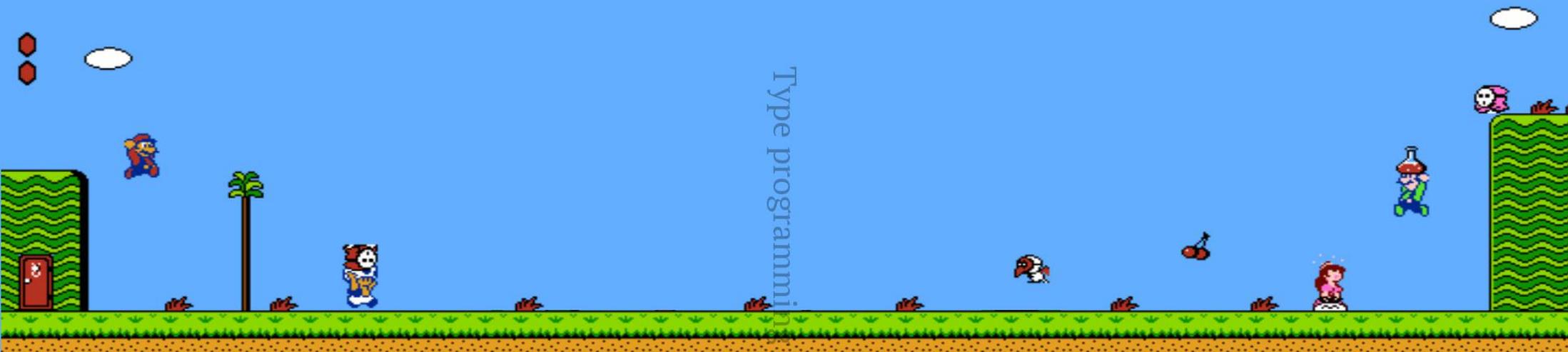
# Retrospective



## Type programming

## Retrospective

Types are not just the shape of my objects

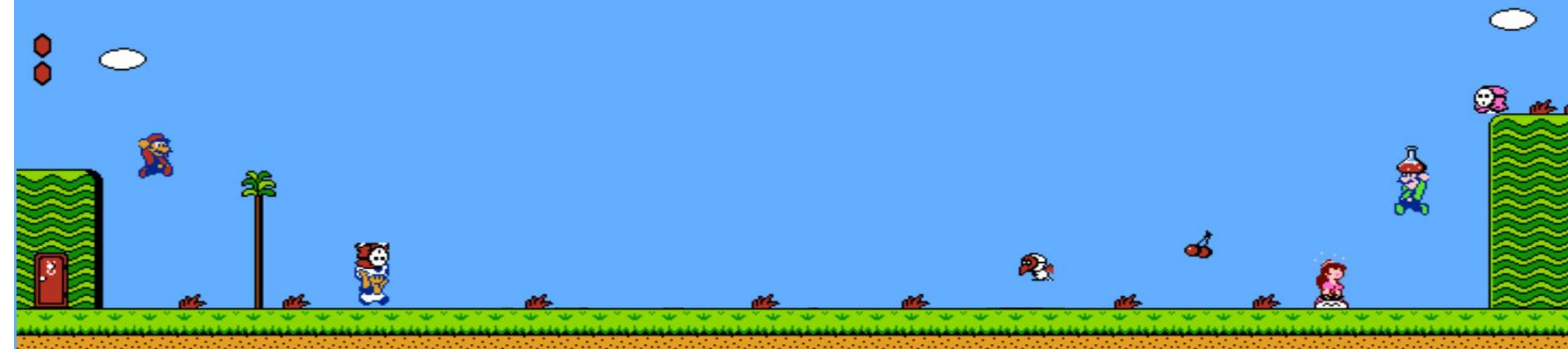


Type programming expedites computation to compile time

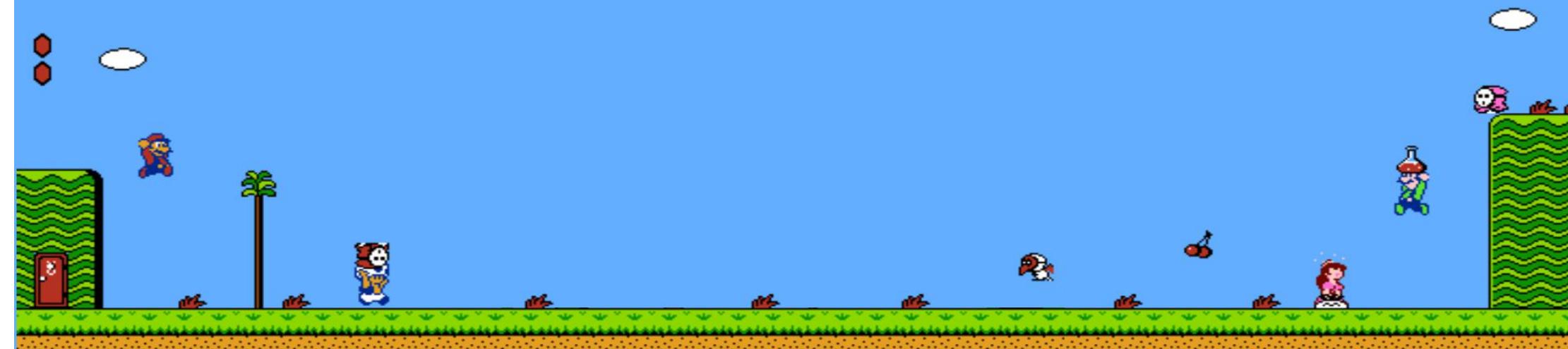
This guarantees I see



This guarantees I see the errors, rather than my users



Type validations propagate through my code base



## Questions?

This guarantees type errors, rather than my users

Type validations propagate through my code base

Type programming expedites compilation to compile time

Types are not just the shape of my objects

## Retrospective

erns?

Type programming expedites compilation to compile time

## Questions?

Types are not just the shapes of my objects

Comments?

Type validations propagate through my code base

This guarantees I see the errors other than my users

   
*Download slides*

   
*Presentation Source*

Retrospective

Type annotations are useful

Initial  
Intermediate  
Advanced



Type annotations are useful



