

λ Programming paradigms

L13: Nominal and structural type system

by Michał Szczepanik

Nominal type system

A nominal type system


(also known as nominative or name-based type system)


is a major class of type system, in which compatibility and equivalence of data types is determined by explicit declarations and/or the name of the types.

Nominal systems are used to determine if types are equivalent, as well as if a type is a subtype of another.

Languages like C++, Java, and Swift have primarily nominal type systems.

Nominal type system

```
class Foo {   
    method(input: string): number { ... }  
}
```

```
class Bar {   
    method(input: string): number { ... }  
}
```

```
let foo: Foo = new Bar(); // Error!!
```

Structural type system

A structural type system

(also known as property-based type system)

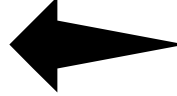
is a major class of type system, in which type compatibility and equivalence are determined by the type's actual structure or definition, and not by other characteristics such as its name or place of declaration.

Structural systems are used to determine if types are equivalent and whether a type is a subtype of another.

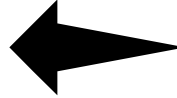
Languages like OCaml, Haskell, and Elm have primarily structural type systems

Structural type system

```
class Foo {  
    method(input: string): number { ... }  
}
```



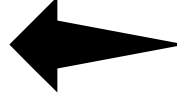
```
class Bar {  
    method(input: string): number { ... }  
}
```



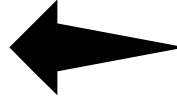
```
let foo: Foo = new Bar(); // Okay!!
```

Structural type system

```
class Foo {  
    method(input: string): number { ... }  
}
```



```
class Bar {  
    method(input: string): boolean { ... }  
}
```



```
let foo: Foo = new Bar(); // Error!!
```

System types

There are also other complex types like objects and functions which can also be either nominal or structural.

Even further, they can be different within the same type system (most of the languages listed before has features of both).

For example:

Flow uses structural typing for objects and functions, but nominal typing for classes.

Functions

```
type FuncType = (input: string) => number;  
function func(input: string): number { ... }  
let test: FuncType = func; // Okay!!
```

In this example a function type alias and an actual function are with the same input and output.

These mean they have the same structure and are equivalent in Flow.

Objects

```
type ObjType = { property: string };  
let obj = { property: "value" };  
let test: ObjType = obj; // Okay!!
```

An object type alias and an actual object with the same properties, only the values are different: string and “value”. Because “value” is a type of string, it is a subtype.

These means that our obj object is a subtype of ObjType.

Classes

```
type Interface = {  
  method(value: string): number;  
}
```

```
class Foo { method(input: string): number {...} }  
class Bar { method(input: string): number {...} }
```

```
let test: Interface = new Foo(); // Okay.  
let test: Interface = new Bar(); // Okay.
```

To use a class structurally they can be mixed with objects as interfaces.

Definition of type in Scala

```
scala> class ClassName
defined class ClassName
scala> trait TraitName
defined trait TraitName
scala> object ObjectName
defined object ObjectName
scala> def f1(x: ClassName) = x
f1: (x: ClassName)ClassName
scala> def f2(x: TraitName) = x
f2: (x: TraitName)TraitName
scala> def f3(x: ObjectName.type) = x
f3: (x: ObjectName.type)ObjectName.type
scala> type ParaIX[Param] = (Int, Param)
defined type alias ParaIX
```

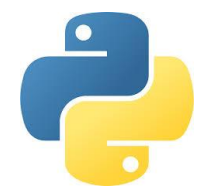
Structural types

```
def fs(x: {}) = x
//fs: (x: AnyRef)AnyRef
class A(var name: String = "A")
//defined class A
var a: {def name: String} = new A
//a: AnyRef{def name: String} = A@52444298
class B(val name: String = "B")
//defined class B
List(a, new B)
//res0: List[Object{def name: String}] = List(A@52444298, B@474f7ab5)
class C(var name: String = "C"){ def draw = println(s"this is an instance of " + this.name) }
//defined class C
val c = new C
//c: C = C@4155a7b1
a = c
//a: AnyRef{def name: String} = C@4155a7b1
a.asInstanceOf[C].draw
//this is an instance of C
```

Duck Typing

Duck Typing is a way of programming in which an object passed into a function or method supports all method signatures and attributes expected of that object at run time.

Python and Ruby support duck typing.



```
class Sparrow:  
    def fly(self):  
        print("Sparrow flying")
```

```
class Airplane:  
    def fly(self):  
        print("Airplane flying")
```

```
class Whale:  
    def swim(self):  
        print("Whale swimming")
```

The **type** of entity is not specified
We expect entity to have a callable named fly at run time

```
def lift_off(entity):  
    entity.fly()
```

```
sparrow = Sparrow()  
airplane = Airplane()  
whale = Whale()
```

```
lift_off(sparrow) # prints `Sparrow flying`  
lift_off(airplane) # prints `Airplane flying`  
lift_off(whale) # Throws the error ``Whale' object has no attribute 'fly'`
```

in Scala

```
class SomeApi {  
  // regular typing method, takes Foo, callers  
  // can only pass a Foo, normal typing  
  def regularMethod(foo: Foo) {  
    // regular dispatching  
    foo.doFoo()  
  }  
  
  // structural typing method, callers can pass  
  // any type that has a "doFoo" method  
  def structuralMethod(foo: { def doFoo(): Unit }) {  
    // structural dispatching (reflection)  
    foo.doFoo()  
  }  
}
```

Classification of typing systems

VS	
Static	Dynamic / Duck
Explicit	Implicit
Nominal	Structural
Strong	Weak

Notions of abstraction

There have always been two notions of abstraction:

- parameterization,
- abstract members.

The first form is typical for functional languages, whereas the second form is typically used in object-oriented languages.

```
// Type parameter version  
trait Collection[T] {  
    // ...  
}
```

```
// Type member version  
trait Collection {  
    type T  
    // ...  
}
```

Type refinement

```
abstract class SmpCell {  
  type T  
  val init: T  
  var value: T  
}
```

```
val cell = new SmpCell {type T = Int; val init = 1; var value = init}
```

The Singleton type

The Singleton type is one of the types in Scala type family; it bridges the abstraction and its inhabitants.

Scala definition:

A singleton type is of the form $p.\mathbf{type}$, where p is a path pointing to a value expected to conform to `scala.AnyRef`. The type denotes the set of values consisting of **null** and the value denoted by p .

```
scala> val s = "hello"
```

```
s: String = hello
```

```
scala> type stype = s.type
```

```
defined type alias stype
```

```
scala> val ss:stype = s
```

```
ss: stype = hello
```

```
scala> val ss:stype = "world"
```

```
<console>:12: error: type mismatch;
```

```
found : String("world")
```

```
required: stype
```

```
(which expands to) s.type
```

```
val ss:stype = "world"
```

```
class A {  
  protected var x = 0  
  def get = x  
  def incr = {x += 1; this}  
}
```

```
class B extends A {  
  def decr = {x -= 1; this}  
}
```

```
val b = new B
```

```
b.incr.decr.get
```

Error:(10, 16) value decr is not a member of A

```
b.incr.decr.get
```

```
b.incr.asInstanceOf[B].decr.get
```

```
def whatType(a: Any) = a match {  
  case _: Byte | _: Short | _: Int | _: Long => "integer number"  
  case _: Float | _: Double => "floating point number"  
  case _: Char => "Char type"  
  case _: Unit => "Unit type"  
  case b: Boolean => s"$b: Boolean"  
  case _ => "reference type"}
```

```
whatType("PP")  
//res0: String = reference type
```

```
whatType(2)  
//res1: String = integer number
```

```
whatType(null)  
//res2: String = reference type
```

Ordered

Scala, defines a Ordered trait that extends the Java Comparable interface.

The Ordered trait defines some additional concrete methods that depend on the compare method.

```
trait Ordered[T] extends Comparable[T] {  
  abstract def compare(that: T): Int  
  def <(that: T): Boolean = // ...  
  def <=(that: T): Boolean = // ...  
  def >(that: T): Boolean = // ...  
  def >=(that: T): Boolean = // ...  
  def compareTo(that: T): Int = // ...  
}
```



```
case class Value(i: Int) extends Ordered[Value] {  
  def compare(that: Value) = this.i - that.i  
}
```

```
scala> val valueList = List(Value(1), Value(2), Value(3), Value(2), Value(1))  
valueList: List[Value] = List(Value(1), Value(2), Value(3), Value(2), Value(1))  
scala> valueList.sorted  
res0: List[Value] = List(Value(1), Value(1), Value(2), Value(2), Value(3))  
scala> valueList.min  
res1: Value = Value(1)
```

!! we can't sort on classes not defining the Ordered trait

Implicit conversions in Scala

Implicit conversions are the set of methods that are applied when an object of wrong type is used. It allows the compiler to automatically convert from one type to another.

Implicit conversions are applied in two conditions:

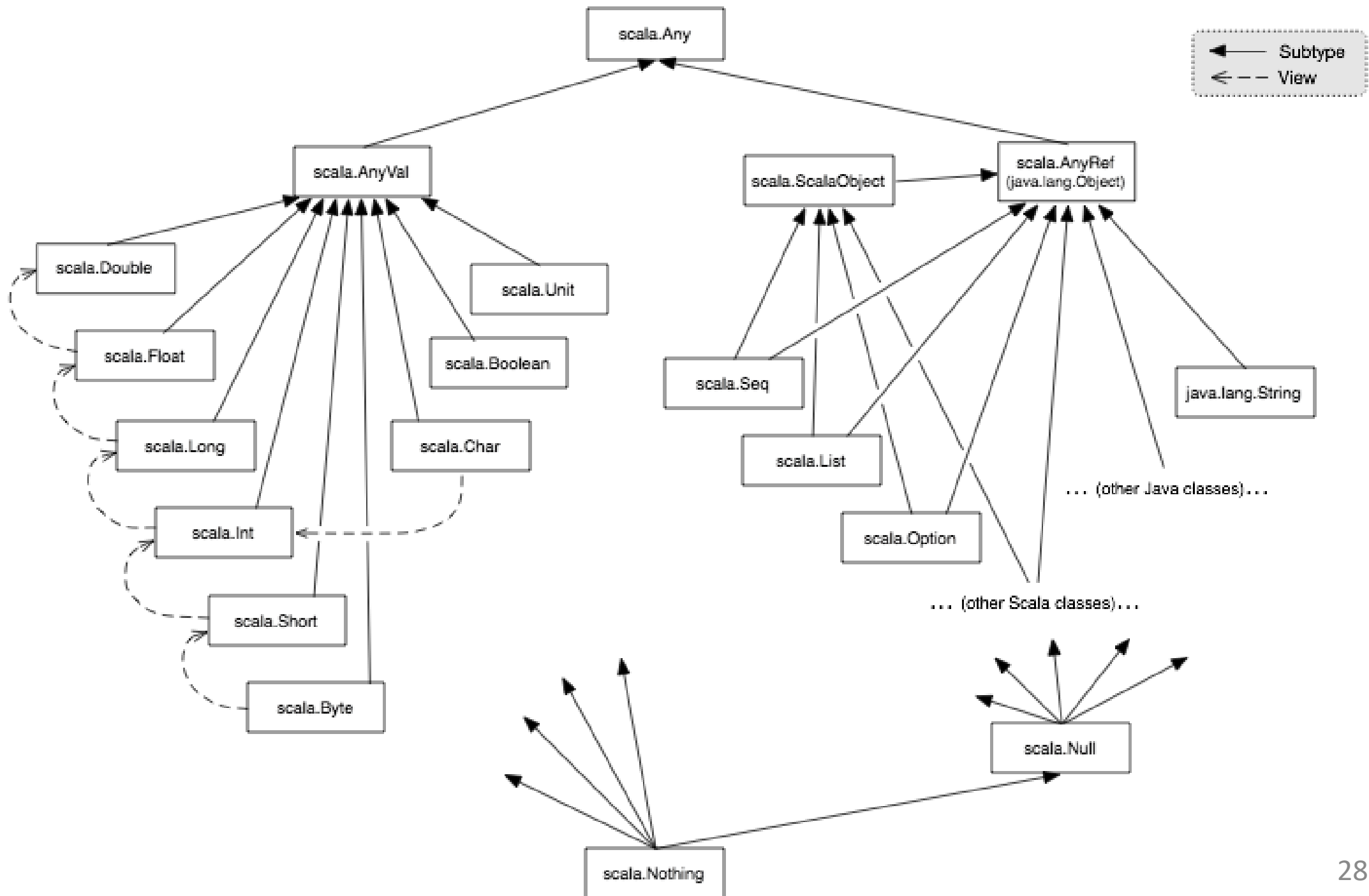
- First, if an expression of type A and S does not match to the expected expression type B.
- Second, in a selection e.m of expression e of type A, if the selector m does not represent a member of A.

In the first condition, a conversion is searched which is appropriate to the expression and whose result type matches to B. In the second condition, a conversion is searched which is appropriate to the expression and whose result carries a member named m.

```
case class A(s: String)
object A
{
  // Using implicitConversions
  implicit def fromString(s: String): A = A(s)
}
```

```
class C
{
  def m1(a: A) = println(a)
  def m(s: String) = m1(s)
}
```

```
object C
{
  def main(args: Array[String])
  {
    var b : A = ("some string")
    println(b)
  }
}
```



Thank you for your attention

michal.szczepanik@pwr.edu.pl