BA-BEAD Big Data Engineering for Analytics

BEAD Workshop Series

W 02 - Scala Practice Workshop

©2015-18 NUS. The contents contained in this document may not be reproduced in any form or by any means, without the written permission of ISS, NUS other than for the purpose for which it has been supplied.

Table of Contents

© 2016-18 NUS. All rights reserved.

Introduction

Before we start writing your data analytics program using Spark and Scala, we will first get familiar with Scala's functional programming concepts, object oriented features and the Scala collection APIs in detail. As a starting point, we will provide a brief introduction to Scala in this workshop. Later in this workshop, we will provide a comparative analysis between Java and Scala. Finally, we will dive into Scala programming with some examples.

Object Oriented Scala

Scala is a very good example of an object-oriented language. To define a type or behaviour for your objects you need to use the notion of classes and traits, which will be explained later, in the next chapter. Scala doesn't support direct multiple inheritances, but to achieve this structure, you need to use Scala's extension of the sub classing and mixing-based composition.

Variables in Scala

Before entering into the depth of OOP features, first, we need to know details about the different types of variables and data types in Scala. To declare a variable in Scala, you need to use var or val keywords. The formal syntax of declaring a variable in Scala is as follows:

```
val or var VariableName : DataType = Initial_Value
```

For example, let's see how can we declare two variables whose data types are explicitly specified as follows:

```
var myVar : Int = 50
val myVal : String = "Hello World! I've started learning Scala."
```

You can even just declare a variable without specifying the DataType. For example, let's see how to declare a variable using val or var, as follows:

```
var myVar = 50
val myVal = "Hello World! I've started learning Scala."
```

There are two types of variables in Scala: mutable and immutable that can be defined as follows:

- **Mutable**: The ones whose values you can change later
- **Immutable**: The ones whose values you cannot change once they have been set

Reference versus value immutability

According to the section earlier, val is used to declare immutable variables, so can we change the values of these variables? Will it be similar to the final keyword in Java? To help us understand more about this, we will use the following code snippet:

```
scala> var testVar = 10
testVar: Int = 10
scala> testVar = testVar + 10
testVar: Int = 20
scala> val testVal = 6
```

© 2016-18 NUS. All rights reserved.

```
testVal: Int = 6
scala> testVal = testVal + 10
<console>:12: error: reassignment to val
testVal = testVal + 10
             ^
scala>
```

If you run the preceding code, an error at compilation time will be noticed, which will tell you that you are trying to reassign to a val variable. In general, mutable variables bring a performance advantage.

Data Types in Scala

Sr.No	Data Type and Description
1	**Byte**: 8 bit signed value. Ranges from -128 to 127
2	**Short**: 16 bit signed value. Ranges -32768 to 32767
3	**Int**: 32 bit signed value. Ranges -2147483648 to 2147483647
4	**Long**: 64 bit signed value. -9223372036854775808 to 9223372036854775807
5	**Float**: 32 bit IEEE 754 single-precision float
6	**Double**: 64 bit IEEE 754 double-precision float
7	**Char**: 16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
8	**String**: A sequence of Chars
9	**Boolean**: Either the literal `true` or the literal `false`
10	**Unit**: Corresponds to no value
11	**Null**: Null or empty reference
12	**Nothing**: The subtype of every other type; includes no values
13	**Any**: The supertype of any type; any object is of type *Any*
14	**AnyRef**: The supertype of any reference type

All the data types listed in the preceding table are objects. However, note that there are no primitive types, as in Java. This means that you can call methods on an `Int`, `Long`, and so on.

Variable initialization

In Scala, it's a good practice to initialize the variables once declared. However, it is to be noted that uninitialized variables aren't necessarily nulls (consider types like `Int`, `Long`, `Double`, `Char`, and so on), and initialized variables aren't necessarily non-null (for example, `val s: String = null`). The actual reasons are that:

- In Scala, types are inferred from the assigned value. This means that a value must be assigned for the compiler to infer the type (how should the compiler consider this code: `val a`? Since a value isn't given, the compiler can't infer the type; since it can't infer the type, it wouldn't know how to initialize it).
- In Scala, most of the time, you'll use `val`. Since these are immutable, you wouldn't be able to declare them and then initialize them afterward.

Although, Scala language requires you to initialize your instance variable before using it, Scala does not provide a default value for your variable. Instead, you have to set up its value manually using the wildcard underscore, which acts like a default value, as follows:

© 2016-18 NUS. All rights reserved.

```
var name:String = _
```

Instead of using the names, such as `val1`, `val2` and so on, you can define your own names:

```
scala> val result = 6 * 5 + 8
result: Int = 38
```

You can use these names in subsequent expressions, as follows:

```
scala> 0.5 * result
res0: Double = 19.0
```

Type annotations

If you use a val or var keyword to declare a variable, its data type will be inferred automatically according to the value that you assigned to this variable. You also have the luxury of explicitly stating the data type of the variable at declaration time.

```
val myVal : Integer = 10
```

Type ascription

Type ascription is used to tell the compiler what types you expect out of an expression, from all possible valid types. Consequently, a type is valid if it respects existing constraints, such as variance and type declarations, and it is either one of the types the expression it applies to "is a," or there's a conversion that applies in scope. So, technically, java.lang.String extends java.lang.Object, therefore any String is also an Object. For example:

```
scala> val s = "Suria R Asai"
s: String = Suria R Asai
scala> val p = s:Object
p: Object = Suria R Asai
scala>
```

Lazy val

The main characteristic of a lazy val is that the bound expression is not evaluated immediately, but once on the first access. Here's where the main difference between val and lazy val lies. When the initial access happens, the expression is evaluated and the result is bound to the identifier, the lazy val. On subsequent access, no further evaluation occurs, instead, the stored result is returned immediately. Let's see an interesting example:

```
scala> lazy val num = 1 / 0
num: Int = <lazy>
```

If you look at the preceding code in Scala REPL, you will notice that the code runs very well without throwing any errors, even though you divided an integer with 0! Let's see a better example:

```
scala> val x = {println("x"); 20}
```

© 2016-18 NUS. All rights reserved.

```
x
x: Int = 20
scala> x
res1: Int = 20
scala>
```

This works and, later on, you can access the value of variable x when required.

Methods in Scala

In this part, we are going to talk about methods in Scala. As you dive into Scala, you'll find that there are lots of ways to define methods in Scala. We will demonstrate them in some of these ways:

```
def min(x1:Int, x2:Int) : Int = {
  if (x1 < x2) x1 else x2
}
```

The preceding declaration of the method takes two variables and returns the smallest among them. In Scala, all the methods must start with the def keyword, which is then followed by a name for this method. Optionally, you can decide not to pass any parameters to the method or even decide not to return anything. You're probably wondering how the smallest value is returned, but we will get to this later. Also, in Scala, you can define methods without curly braces:

```
def min(x1:Int, x2:Int):Int= if (x1 < x2) x1 else x2
```

If your method has a small body, you can declare your method like this. Otherwise, it's preferred to use the curly braces in order to avoid confusion. As mentioned earlier, you can pass no parameters to the method if needed:

```
def getPiValue(): Double = 3.14159
```

A method with or without parentheses signals the absence or presence of a side effect. Moreover, it has a deep connection with the uniform access principle. Thus, you can also avoid the braces as follows:

```
def getValueOfPi : Double = 3.14159
```

There are also some methods which return the value by explicitly mentioning the return types. For example:

```
def sayHello(person :String) = "Hello " + person + "!"
```

It should be mentioned that the preceding code works due to the Scala compiler, which is able to infer the return type, just as with values and variables.

This will return Hello concatenated with the passed person name. For example:

```
scala> def sayHello(person :String) = "Hello " + person + "!"
sayHello: (person: String)String
scala> sayHello("Suria")
res2: String = Hello Suria!
scala>
```

© 2016-18 NUS. All rights reserved.

Before learning how a Scala method returns a value, let's recap the structure of a method in Scala:

```
def functionName ([list of parameters]) : [return type] = {
  function body
  value_to_return
}
```

For the preceding syntax, the return type could be any valid Scala data type and a list of parameters will be a list of variables separated by a comma and a list of parameters and return type is optional. Now, let's define a method that adds two positive integers and returns the result, which is also an integer value:

```
scala> def addInt( x:Int, y:Int ) : Int = {
     |       var sum:Int = 0
     |       sum = x + y
     |       sum
     |    }
addInt: (x: Int, y: Int)Int

scala> addInt(20, 34)
res3: Int = 54
scala>
```

If you now call the preceding method from the main() method with the real values, such as addInt(10, 30), the method will return an integer value sum, which is equal to 40. As using the keyword return is optional, the Scala compiler is designed such that the last assignment will be returned with the absence of the return keyword. As in this situation, the greater value will be returned:

```
scala> def max(x1 : Int , x2: Int)  = {
     |     if (x1>x2) x1 else x2
     | }
max: (x1: Int, x2: Int)Int
scala> max(12, 27)
res4: Int = 27
scala>
```

Classes in Scala

Classes are considered as a blueprint and then you instantiate this class in order to create something that will actually be represented in memory. They can contain methods, values, variables, types, objects, traits, and classes which are collectively called **members**. Let's demonstrate this with the following example:

```
class Animal {
  var animalName = null
  var animalAge = -1
  def setAnimalName (animalName:String)  {
    this.animalName = animalName
  }
```

© 2016-18 NUS. All rights reserved.

```
  def setAnaimalAge (animalAge:Int) {
    this.animalAge = animalAge
  }
  def getAnimalName () : String = {
    animalName
  }
  def getAnimalAge () : Int = {
    animalAge
  }
}
```

Objects in Scala

An object in Scala has a slightly different meaning than the traditional OOP one, and this difference should be explained. In particular, in OOP, an object is an instance of a class, while in Scala, anything that is declared as an object cannot be instantiated! The object is a keyword in Scala. The basic syntax for declaring an object in Scala is as follows:

```
object <identifier> [extends <identifier>] [{ fields, methods, and
classes }]
```

To understand the preceding syntax, let's revisit the hello world program:

```
object HelloWorld {
  def main(args : Array[String]){
    println("Hello world!")
  }
}
```

This hello world example is pretty similar to the Java ones. The only big difference is that the main method is not inside a class, but instead it's inside an object. In Scala, the keyword object can mean two different things:

- As in OOP, an object can represent an instance of a class
- A keyword for depicting a very different type of instance object called **Singleton**

When a singleton object is named the same as a class, it is called a companion object. A companion object must be defined inside the same source file as the class. Let's demonstrate this with the example here:

```
class Animal {
  var animalName:String  = "notset"
  def setAnimalName(name: String) {
    animalName = name
  }
  def getAnimalName: String = {
    animalName
  }
  def isAnimalNameSet: Boolean = {
    if (getAnimalName == "notset") false else true
  }
}
```

The following is the way that you will call methods through the companion object (preferably with the same name - that is, Animal):

```
object Animal{
  def main(args: Array[String]): Unit= {
    val obj: Animal = new Animal
    var flag:Boolean  = false
    obj.setAnimalName("dog")
    flag = obj.isAnimalNameSet
    println(flag)  // prints true

    obj.setAnimalName("notset")
    flag = obj.isAnimalNameSet
    println(flag)   // prints false
  }
}
```

Access and Visibility

Modifier	**Class**	**Companion Object**	**Package**	**Subclass**	**Project**
Default/No modifier	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No	No
Private	Yes	Yes	No	No	No

Constructors

There are two types of constructors in Scala - primary and auxiliary constructors. The primary constructor is the class's body, and it's parameter list appears right after the class name. For example, the following code segment describes the way to use the primary constructor in Scala:

```
class Animal (animalName:String, animalAge:Int) {
  def getAnimalName () : String = {
    animalName
  }
  def getAnimalAge () : Int = {
    animalAge
  }
}
```

Now, to use the preceding constructor, this implementation is similar to the previous one, except there are no setters and getters. Instead, we can get the animal name and age, as here:

```
object RunAnimalExample extends App{
  val animalObj = new animal("Cat",-1)
  println(animalObj.getAnimalName)
```

© 2016-18 NUS. All rights reserved.

```
  println(animalObj.getAnimalAge)
}
```

Parameters are given in the class definition time to represent constructors. If we declare a constructor, then we cannot create a class without providing the default values of the parameters that are specified in the constructor. Moreover, Scala allows the instantiation of an object without providing the necessary parameters to its constructor: this happens when all constructor arguments have a default value defined.

Although there is a constraint for using the auxiliary constructors, we are free to add as many additional auxiliary constructors as we want. An auxiliary constructor must, on the first line of its body, call either another auxiliary constructor that has been declared before it, or the primary constructor. To obey this rule, each auxiliary constructor will, either directly or indirectly, end up invoking the primary constructor.

For example, the following code segment demonstrates the use of the auxiliary constructor in Scala:

```
class Hello(primaryMessage: String, secondaryMessage: String) {
  def this(primaryMessage: String) = this(primaryMessage, "")
  // auxilary constructor
  def sayHello() = println(primaryMessage + secondaryMessage)
}
object Constructors {
  def main(args: Array[String]): Unit = {
    val hello = new Hello("Hello world!", " I'm in a trouble,
                          please help me out.")
    hello.sayHello()
  }
}
```

A trait syntax

You need to use the trait keyword in order to declare a trait and it should be followed by the trait name and body:

```
trait Animal {
  val age : Int
  val gender : String
  val origin : String
 }
```

In order to extend traits or classes, you need to use the extend keyword. Traits cannot be instantiated because it may contain unimplemented methods. So, it's necessary to implement the abstract members in the trait:

```
trait Cat extends Animal{ }
```

© 2016-18 NUS. All rights reserved.

A value class is not allowed to extend traits. To permit value classes to extend traits, universal traits are introduced, which extends for Any. For example, suppose that we have the following trait defined:

```
trait EqualityChecking {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

Now, to extend the preceding trait in Scala using the universal trait, we follow the following code segment:

```
trait EqualityPrinter extends Any {
  def print(): Unit = println(this)
}
```

Abstract classes and the override

If you want to override a concrete method from the superclass, the override modifier is necessary. However, if you are implementing an abstract method, it is not strictly necessary to add the override modifier. Scala uses the override keyword to override a method from a parent class. For example, suppose you have the following abstract class and a method printContents() to print your message on the console:

```
abstract class MyWriter {
   var message: String = "null"
   def setMessage(message: String):Unit
   def printMessage():Unit
}
```

Now, add a concrete implementation of the preceding abstract class to print the contents on the console as follows:

```
class ConsolePrinter extends MyWriter
{
     def setMessage(contents: String):Unit= {
       this.message = contents
    }
    def printMessage():Unit= {
        println(message)
    }
}
```

Secondly, if you want to create a trait to modify the behavior of the preceding concrete class, as follows:

```
trait lowerCase extends MyWriter {
    abstract override def setMessage(contents: String) =
printMessage()
}
```

© 2016-18 NUS. All rights reserved.

If you look at the preceding code segment carefully, you will find two modifiers (that is, abstract and override). Now, with the preceding setting, you can do the following to use the preceding class:

```
val printer:ConsolePrinter = new ConsolePrinter()
printer.setMessage("Hello! world!")
printer.printMessage()
```

In summary, we can add an override keyword in front of the method to work as expected.

Functional Scala

Functional programming treats functions like first-class citizens. In Scala, this is achieved with syntactic sugar and objects that extend traits (like Function2), but this is how functional programming is achieved in Scala. Also, Scala defines a simple and easy way to define anonymous functions (functions without names). It also supports higher-order functions and it allows nested functions. Also, it helps you to code in an immutable way, and by this, you can easily apply it to parallelism with synchronization and concurrency.

In Scala, functions are named, reusable expressions. They may be parameterized and they may return a value, but neither of these features are required. These features are, however, useful for ensuring maximum reusability and composability. They will also help you write shorter, more readable, and more stable applications. Using parameterized functions, you can normalize duplicated code, simplifying your logic and making it more discoverable. Testing your code becomes easier, because normalized and parameterized logic is easier to test than denormalized logic repeated throughout your code.

Even greater benefits may come from following standard functional programming methodology and building pure functions when possible. In functional programming a pure function is one that:

- Has one or more input parameters
- Performs calculations using only the input parameters
- Returns a value
- Always returns the same value for the same input
- Does not use or affect any data outside the function
- Is not affected by any data outside the function

Pure functions are essentially equivalent to functions in mathematics, where the definition is a calculation derived only from the input parameters, and are the building blocks for programs in functional programming. They are more stable than functions that do not meet these requirements because they are stateless and orthogonal to external data such as files, databases, sockets, global variables, or other shared data. In essence, they are uncorruptible and noncorrupting expressions of pure logic.

© 2016-18 NUS. All rights reserved.

Syntax: Defining an Input-less Function

```
def <identifier> = <expression>
```

At its most basic, a Scala function is a named wrapper for an expression. When you need a function to format the current data, check a remote service for new data, or just to return a fixed value, this is the format for you. Here is an example of defining and invoking input-less functions:

```
scala> def hi = "hi"
hi: String
scala> hi
res0: String = hi
```

The return type of functions, as with values and variables, are present even if they are not explicitly defined. And like values and variables, functions are easier to read with explicit types.

Syntax: Defining a Function with a Return Type

```
def <identifier>: <type> = <expression>
```

This function definition is also input-less, but it demonstrates the “colon-and-type” format from value and variable definitions for function definitions. Here’s the “hi” function again with an explicit type for better readability:

```
scala> def hi: String = "hi"
hi: String
```

Now we’re ready to look at a full function definition.

Syntax: Defining a Function

```
def <identifier>(<identifier>: <type>[, ... ]): <type> =
<expression>
```

Let’s try creating a function that performs an essential mathematical operation:

```
scala> def multiplier(x: Int, y: Int): Int = { x * y }
multiplier: (x: Int, y: Int)Int
scala> multiplier(6, 7)
res0: Int = 42
```

The body of these functions consists essentially of expressions or expression blocks, where the final line becomes the return value of the expression and thus the function. While I do recommend continuing this practice for functions, there are times when you need to exit and return a value before the end of the function’s expression block. You can use the return keyword to specify a function’s return value explicitly and exit the function.

A common use of an early function exit is to stop further execution in the case of invalid or abnormal input values. For example, this “trim” function validates that the input value is nonnull before calling the JVM String’s “trim” method:

```
scala> def safeTrim(s: String): String = {
 |   if (s == null) return null
|   s.trim()
```

© 2016-18 NUS. All rights reserved.

```
| }
safeTrim: (s: String)String
```

You should now have a basic understanding of how to define and invoke functions in Scala.

Functions with Empty Parentheses

An alternate way to define and invoke an input-less function (one which has no input parameters) is with empty parentheses. You might find this style preferable because it clearly distinguishes the function from a value.

Syntax: Defining a Function with Empty Parentheses

```
def <identifier>()[: <type>] = <expression>
```

You can invoke such a function using empty parentheses as well, or choose to leave them off:

```
scala> def hi(): String = "hi"
hi: ()String
scala> hi()
res1: String = hi
scala> hi
res2: String = hi
```

The reverse is not true, however. Scala does not allow a function that was defined without parentheses to be invoked with them. This rule prevents confusion from invoking a function without parentheses versus invoking the return value of that function as a function.

Recursive Functions

A recursive function is one that may invoke itself, preferably with some type of parameter or external condition that will be checked to avoid an infinite loop of function invocation. Recursive functions are very popular in functional programming because they offer a way to iterate over data structures or calculations without using mutable data, because each function call has its own stack for storing function parameters.

Here's an example of a recursive function that raises an integer by a given positive exponent:

```
scala> def power(x: Int, n: Int): Long = {
   |   if (n >= 1) x * power(x, n-1)
   |   else 1
   | }
power: (x: Int, n: Int)Long
scala> power(2, 8)
res6: Long = 256
scala> power(2, 1)
res7: Long = 2
scala> power(2, 0)
res8: Long = 1
```

One problem with using recursive functions is running into the dreaded "Stack Overflow" error, where invoking a recursive function too many times eventually uses up all of the

© 2016-18 NUS. All rights reserved.

allocated stack space. To prevent this scenario, the Scala compiler can optimize some recursive functions with tail-recursion so that recursive calls do not use additional stack space.

Nested Functions

Functions are named, parameterized expression blocks and expression blocks are nestable, so it should be no great surprise that functions are themselves nestable.

There are times when you have logic that needs to be repeated inside a method, but would not benefit from being extrapolated to an external method. In these cases defining an internal function inside another function, to only be used in that function, may be worthwhile.

Let's have a look at a method that takes three integers and returns the one with the highest value:

```
scala> def max(a: Int, b: Int, c: Int) = {
     |   def max(x: Int, y: Int) = if (x > y) x else y
     |   max(a, max(b, c))
     | }
max: (a: Int, b: Int, c: Int)Int

scala> max(42, 181, 19)
res10: Int = 181
```

The logic inside the max(Int, Int) nested function was defined once but used twice inside the outer function, making it possible to reduce duplicated logic and simplify the overall function.

Calling Functions with Named Parameters

The convention for calling functions is that the parameters are specified in the order in which they are originally defined. However, in Scala you can call parameters by name, making it possible to specify them out of order.

Syntax: Specifying a Parameter by Name

```
<function name>(<parameter> = <value>)
```

In this example, a simple two-parameter function is invoked twice, first using the convention of specifying parameters by their order and then by assigning values by parameter name:

```
scala> def greet(prefix: String, name: String) = s"$prefix $name"
greet: (prefix: String, name: String)String
scala> val greeting1 = greet("Ms", "Brown")
greeting1: String = Ms Brown
scala> val greeting2 = greet(name = "Brown", prefix = "Mr")
greeting2: String = Mr Brown
```

Methods and Operators

Until this point we have been discussing the use of functions without reference to where they will actually be used. Functions on their own, as defined in the REPL, are helpful for learning

© 2016-18 NUS. All rights reserved.

the core concepts. However, in practice they will typically exist in objects and act on data from the object, so a more appropriate term for them will often be "methods."

A method is a function defined in a class and available from any instance of the class. The standard way to invoke methods in Scala (as in Java and Ruby) is with infix dot notation, where the method name is prefixed by the name of its instance and the dot (.) separator.

Syntax: Invoking a Method with Infix Dot Notation

```
<class instance>.<method>[(<parameters>)]
```

Let's try this out by calling one of the many useful methods on the String type:

```
scala> val s = "vacation.jpg"
s: String = vacation.jpg
scala> val isJPEG = s.endsWith(".jpg")
isJPEG: Boolean = true
```

If it isn't clear, the value s is an instance of type String, and the String class has a method called endsWith(). In the future we'll refer to methods using the full class name, like String.endsWith(), even though you typically invoke them with the instance name, not the type name.

Higher-Order Functions

We have already defined values that have a function type. A *higher-order* function is a function that has a value with a function type as an input parameter or return value.

Here's a good use case for a higher-order function: calling other functions that act on a String, but only if the input String is not null. Adding this check can prevent a NullPointerException in the JVM by avoiding a method call on null:

```
scala> def safeStringOp(s: String, f: String => String) = {
     |   if (s != null) f(s) else s
     | }
safeStringOp: (s: String, f: String => String)String

scala> def reverser(s: String) = s.reverse
reverser: (s: String)String

scala> safeStringOp(null, reverser)
res4: String = null

scala> safeStringOp("Ready", reverser)
res5: String = ydaeR
```

The call with "null" safely returned the same value back, whereas the call with a valid String returned the reverse of the input value.

This example demonstrated how to pass an existing function as a parameter to a higher-order function.

© 2016-18 NUS. All rights reserved.

Exercises

1. Write a function that computes the area of a circle given its radius.
2. Provide an alternate form of the function in exercise 1 that takes the radius as a `String`. What happens if your function is invoked with an empty `String` ?
3. Write a recursive function that prints the values from 5 to 50 by fives, without using `for` or `while` loops. Can you make it tail-recursive?
4. Write a function that takes a milliseconds value and returns a string describing the value in days, hours, minutes, and seconds. What's the optimal type for the input value?
5. Write a function that calculates the first value raised to the exponent of the second value. Try writing this first using `math.pow`, then with your own calculation. Did you implement it with variables? Is there a solution available that only uses immutable data? Did you choose a numeric type that is large enough for your uses?
6. Write a function that calculates the difference between a pair of 2D points (x and y) and returns the result as a point. Hint: this would be a good use for tuples (see Tuples).
7. Write a function that takes a 2-sized tuple and returns it with the `Int` value (if included) in the first position. Hint: this would be a good use for type parameters and the `isInstanceOf` type operation.
8. Write a function that takes a 3-sized tuple and returns a 6-sized tuple, with each original parameter followed by its `String` representation. For example, invoking the function with `(true, 22.25, "yes")` should return `(true, "true", 22.5, "22.5", "yes", "yes")`. Can you ensure that tuples of all possible types are compatible with your function? When you invoke this function, can you do so with explicit types not only in the function result but in the value that you use to store the result?

© 2016-18 NUS. All rights reserved.