







### Example

### Listing: Mutable

### Listing: Immutable





## Build-in Types

## Hierarchy

see: <https://docs.scala-lang.org/tour/unified-types.html>

```
1 var tInt : Int = 8
2 // error
3 // tInt = "eight"
4 var tString : String = "eight"
5 //error
6 // tString = 8
7 var tAny : Any = "eight"
8 tAny = 8
```

### Listing: Type Hierarchy





## Operator

`+` `-` `*` `/` `%` arithmetic: add, subtract, multiply, divide, modules

==! =><>=<= relational:

**&& || !** logical: and, or, not

- & | bitwise: and, or

<< >> >>> bitwise: left shift, right shift,

`= + = - = * = / = % => > = < < = = | =` 32-bit IEEE 754  
single-precision float







## Control Structure

## if or if-else

```
1 import scala.io.StdIn._
2
3 def numberChecking() : Unit =
4     val inputNum = readLine("Input a number: ")
5     if inputNum.toInt % 2 == 0 then
6         println("Your number is even")
7     else
8         println("Your number is odd")
```

### Listing: example of if-else

## Control Structure

## while-do

```
1 while
2     expression1 : Boolean
3 do
4     expression2 : A
```

### Listing: while-do

expression1 evaluate to Boolean

**expression2** sequence of expression that can evaluate to any value



## Control Structure

## while-do

```
1 def print1to20() : Unit =  
2     var number = 1  
3     while  
4         number <= 20  
5     do  
6         println(number)  
7         number += 1
```

### Listing: example of while-do



## Control Structure

for-do

```
1 for expression1 do
2     expression2
3     ...
4     expressionN
```

### Listing: for-do

expression1 evaluate to Boolean

**expression2** sequence of expression that can evaluate to any value



## Control Structure

## for-do

```
1 def triangleStar(): Unit =
2   for(a <- 1 to 5) do
3     for(b <- 1 to 5) do
4       if(b <= a) then
5         print("*")
6       println("")
```

### Listing: example for-do







# Control Structure

## for-do with Guards

```
1 for
2   forExp1
3   ...
4   forExpN
5   ifExp1 : Boolean
6   ...
7   ifExpN : Boolean
8 do
9   expBody
```

Listing: for-do with guards format

**forExp1 ... forExpN** expressions for iteration, it can hold multiple expressions

**ifExp1 ... ifExpN** expression for filtering using *if*, it can hold multiple filtering

**expBody** body of expression



# Control Structure

## for-do with Guards

```
1 for(i <- 1 to 10) do
2   if(i % 2 == 0) then
3     println(i)
```

Listing: filter inside the body

```
1 for
2   i <- 1 to 10
3   if i % 2 == 0
4 do
5   println(i)
```

Listing: filter inside the for-expression



# Control Structure

## for-do with Guards

```
1 for
2   i <- 1 to 10
3   if i % 2 == 0
4   if i > 6
5 do
6   println(i)
```

Listing: for-do with guards example1

More than one filter expression execution of filter will do sequentially from the first to the last expression.

How about more than one iteration's expression?





# Control Structure

## for-do comprehensions

### for comprehensions

a syntactic sugar for composition of multiple monadic operation.

```
1 for
2   a <- exp1
3   b <- exp2
4   c <- exp3
5 do
6   expBody
```

Listing: for-do syntactic sugar

```
1 exp1.foreach(a => exp2.foreach(b => exp3.foreach(c =>
   expBody)))
```

Listing: for-do comprehension



# Control Structure

## for-do comprehensions

```
1 for
2   x <- 1 to 10
3 do
4   println(x)
5
6 for
7   x <- List("Hello", "World")
8 do
9   println(x)
```

Listing: for-do syntactic sugar - example

```
1 for
2   x <- List("Hello", "World")
3 do
4   println(x)
5
6 (1 to 10).foreach(x => println(x))
```

Listing: for-do comprehension - example



# Control Structure

## for-do comprehensions

```
1 for
2   x <- 1 to 10
3 do
4   for
5     y <- List("Hello", "World")
6 do
7   println(s"x = $x, y = $y")
```

Listing: for-do nested - example

```
1 for
2   x <- 1 to 10
3   y <- List("Hello", "World")
4 do
5   println(s"x = $x, y = $y")
```

Listing: for-do nested - example

```
1 (1 to 10).foreach(x => List("Hello", "World").foreach(y =>
   println(s"x = $x, y = $y")))
```

Listing: for-do nested comprehension - example



# Control Structure

## for-do comprehensions

Is any task that can do using nested-for but can't use for-comprehension? Give a simple example!





# Control Structure

## for-do comprehensions

```
1 for
2   x <- 1 to 10
3 do
4   println(x)
5   for
6     y <- List("Hello", "World")
7   do
8     println(y)
```

Listing: for-do nested with a specific task per iteration

# Control Structure

for-yield

Looping for-do with Output Buffer-yield



# Control Structure

## for-yield

For each iteration of your for loop, yield generates a value which will be remembered. It's like the for loop has a buffer you can't see, and for each iteration of your for-loop, another item is added to that buffer. When your for-loop finishes running, it will return this collection of all the yielded values. The type of the collection that is returned is the same type that you were iterating over, so a Map yields a Map, a List yields a List, and so on.

— Alexander —



# Control Structure

## for-yield

```
1 for
2   forExp1
3   ...
4   forExpN
5   ifExp1 : Boolean
6   ...
7   ifExpN : Boolean
8 yield
9   yBody
```

Listing: for-yield

**forExp1 ... forExpN** expressions for iteration, it can hold multiple expressions.

**ifExp1 ... ifExpN** expression for filtering using *if*, it can hold multiple filtering.

**yBody** body of expression.



## Control Structure

for-**yield**

```
1 var yA = for i <- 1 to 10 yield i
2
3 def foo(n: Int, v: Int) =
4   for i <- 0 until n
5     j <- 0 until n if i + j == v
6   yield i+j
7
8 var yB =
9   for
10     i <- 1 to 10
11     y <- List("Hello", "World")
12   yield
13     println("yield")
14     (i,y)
```

### Listing: for-*yield* example



# Control Structure

## match

Scala has a match expression, which in its most basic use is like a Java switch statement. –scala3 book

```
1 vPM : Matchable match
2   case val1 => exp1 : T
3   case val2 => exp2 : T
4   case valn => expn : T
5   case _   => exp_  : T
```

Listing: match



# Control Structure

match

```
1 def singleDigitToText (d: Int) : String = d match
2   case 1 => "one"
3   case 2 => "two"
4   case 3 => "three"
5   // ... to 9
6   case _ => "unknown"
```

Listing: matching on value - ex-1



## Control Structure

match

```
1 def checkParamType (p: Any) : String = p match
2   case (_: Int) => "Integer"
3   case (_: Char) => "Character"
4   case (_: String) => "String"
5   case (_: Double) => "Double"
6   // ... to all possible types
7   case _ => "unknown"
```

### Listing: matching on type - ex-2





## Control Structure

match

```
1 def checkParamType2 (p: Any) : (Any, String) = p match
2   case (v: Int) => (v, "Integer")
3   case (v: Char) => (v, "Character")
4   case (v: String) => (v, "String")
5   case (v: Double) => (v, "Double")
6   // ... to all possible types
7   case _ => (p, "unknown")
```

### Listing: matching on type - ex-3



## Control Structure

for-yield

```

1 def checkInRange(v: Int) = v match
2   case n if 0 to 10 contains n => s"$n is smaller than 10"
3   case x if x >= 10 && x < 20 => s"$x is in between 10 and
4     19"
5   case 20 | 21 | 22 | 23 | 24 => s"$v is in between 20 and
6     24"
7   case _ => s"$v is greather than 19"
8
9 def factorial(v: Int, acc: Int) : Int = v match
10  case x if x < 1 => acc
11  case x => factorial((x-1), (acc * x))

```

Listing: matching with guards - ex-4



## Control Structure

match

```

1 // Higher-order function
2 def rFold[A](lA: List[A], idA: A, optr: (A,A) => A) : A = lA
   match
3   case Nil    => idA
4   case (x :: Nil) => optr(x, idA)
5   case (x :: xs)  => optr(x, rFold(xs, idA, optr))
6
7 val ex1 = rFold(List(1,2,3,4), 0, (a,b) => a + b)
8 val ex2 = rFold(List("Hello", "World!"), "", (a,b) => a + b)

```

**Listing:** matching on list constructor - ex-5

## Control Structure

match

```
1 // matching on object
2 case class Person(id: Int)
3
4 def mapIdToName(p: Person) : String = p match {
5     case Person(7) => "James Bond"
6     case Person(6) => "Hitman"
7     case _         => "Unknown id"
8 }
9 mapIdToName(new Person(7))
```

### Listing: matching on object - ex-6

## Control Structure

match

```
1 // Mini Calculator Language
2 sealed trait Exp
3 case class Num(n: Int) extends Exp
4 case class Add(lhs: Exp, rhs: Exp) extends Exp
5 case class Mul(lhs: Exp, rhs: Exp) extends Exp
6
7 def eval (exp: Exp) : Int = exp match
8   case Num(n) => n
9   case Add(lhs, rhs) => eval(lhs) + eval(rhs)
10  case Mul(lhs, rhs) => eval(lhs) * eval(rhs)
11  case _ => sys.error("Unknown expression")
12
13 eval(Num(3))
14 eval(Add(Num(7), Num(8)))
```

### Listing: matching on object - ex-7



## try-catch-finally

```

1 def divX(vD : Double, divD: Double) = divD match
2   case 0 => throw new ArithmeticException("Error! divider
      is zero")
3   case _ => vD / divD
4
5 def testDiv =
6   try
7     divX(3,0)
8   catch
9     case ex: ArithmeticException => ex.printStackTrace()
10  finally
11    println("the last step")

```

### Listing: try-catch-finally example

