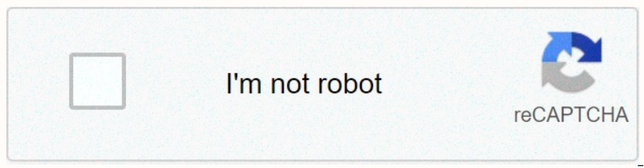

Continue

Java ternary operator throw exception

We often discuss Java restrictions on IRC and try to come up with (sometimes stupid) solutions. Unfortunately over time it is often easy to forget the result, and lose code fragments. So I thought I would start blogging some of them so I wouldn't lose them, and other people might suggest other ways to do things we overlooked. This particular issue occurs when you want to assign the result of a method that can make an exception to a final variable. For example: end customer c; try { c = getCustomer(id); } catch (CustomerNotFoundException e) { c = createNewCustomer(); } end customer c; try { c = getCustomer(id); } catch (CustomerNotFoundException e) { c = createNewCustomer(); } This will not compile with variable c may already have been assigned. Of course making c not final would solve the problem, but that's not fun. If we didn't use exceptions, Java provides a useful ternary operator ?: with which we can do things like: final Customer c = customerExists(id) ? getCustomer(id) : createNewCustomer();final Customer c = customerExists(id) ? getCustomer(id) : createNewCustomer(); That's nice and clean, but means getCustomer is going to have to return null or throw out a RuntimeException in case there's no matching customer, which is undesirable. CustomerExists() can also be expensive. We may also use something along the lines of the latter option<Customer> c = getCustomer();last option<Customer> c = getCustomer(); However, both alternatives require changing the API you consume and avoiding exceptions. It would be nice if there was an equivalent of ?: for try/catch so that you could assign the result to a final variable. The best I can manage in Java is below, can anyone do better? import java.lang.reflect.ParameterizedType; abstract class TryCatch<T, u= extends= exception=> { public T value() { try {return Try(); } catch (Exception e) { if (getTypeOfU().isAssignableFrom(e.getClass()) { return Catch(); else { throw new RuntimeException(e); } } @SuppressWarnings(uncontrolled) private class<U> getTypeOfU() { return (Class<U>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[1]; } public abstract T Try() throws You; public abstract T Catch(); } //Example public class Main { private static CustomerRepo repo = new CustomerRepo(); public static static void main(String[] args) { final Customer c = new TryCatch<Customer, customernotfoundexception=>() { public Customer Try() throws CustomerNotFoundException { System.out.println(in try); return repo.getCustomer(1); } public customer catch() { System.out.println(in catch); repo.createCustomer(); } value(); } } class { public Customer getCustomer(int id) gooit CustomerNotFoundException { gooi nieuwe CustomerNotFoundException(); } openbare klant</Customer,> </U> </U> </T,> </Customer> </Customer> </Customer> { new customer(); } return } Customer class { } CustomerNotFoundException class extends Exception { }import java.lang.reflect.ParameterizedType; abstracte klasse TryCatch<T, u= extends= exception=> { public T value() { try {return Try(); } catch (Exception e) { if (getTypeOfU().isAssignableFrom(e.getClass())) { return Catch(); else { throw new RuntimeException(e); } @SuppressWarnings 50(unchecked) private Class<U> getTypeOfU() { return (Class<U>) (((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[1]; } openbare abstracte T Try() gooit U; openbare abstracte T Catch(); } //Voorbeeld openbare klasse Main { private statische CustomerRepo repo = nieuwe CustomerRepo(); openbare statische void main(String[] args) { final Customer c = new TryCatch<Customer, customernotfoundexception=>() { public Customer Try() gooit CustomerNotFoundException { System.out.println(in try); return repo.getCustomer(1); } public Customer Catch() { System.out.println(in catch); return repo.createCustomer(); } }. waarde(); } } klasse CustomerRepo { public Customer getCustomer(int id) gooit CustomerNotFoundException { gooi nieuwe CustomerNotFoundException(); } openbare Klant maaktCustomer() { return new Customer(); } klasse Klant { } klasse CustomerNotFoundception breidt Exception uit { } In C# lopen we niet tegen hetzelfde probleem aan omdat readonly veel minder nuttig is dan de finale van Java. However, if we wanted to try/catch at the same time we could do a little better. Here is an alternative in C #: using System; Public Class Example { Public Static Void Main() { Example t = New Example(); t.Foo(); } Public Void Foo() { String Result1 = this. Try(() => GetBar(true)). Catch<BarException>(() => Caught a BarException); String result2 = this. Try(() => GetBar(false)). Catch<BarException>(() => Caught a BarException); Console.WriteLine(result1); Console.WriteLine(result2); } Public String GetBar(bool blows) { as (strokes) return Success!; otherwise throw new BarException(); } public class BarException : Exception {} Public Class Tryer<TResult> { private readonly Func<TResult> toTry; internal Tryer(Func<TResult> toTry) { this.toTry = toTry; } public TResult Catch<TException>(Func<TResult> whenCaught) where TException : Exception { try { return toTry(); } catch (TException) { return whenCaught(); } } namespace System { public static class Provides { public static Tryer<TResult> Try<T,TResult>(this T other, Func<TResult> toTry) { return new Tryer<TResult>(toTry); } } Follow @benjiweber Tags: Java Julia offers a variety of control flow constructs:The first five control flow mechanisms are standard for high-level programming languages. Tasks are not as standard: they provide a non-local control flow, making it possible to switch between temporarily suspended calculations. This is a powerful construction: both exception handling and cooperative multitasking are implemented in Julia with tasks. Daily programming does not require direct use of tasks, but certain problems can be solved much more easily by using tasks. Compound expressionsSome times it is useful to have a single expression that evaluates different subexpressions in order and returns the value of the last subexpression as value. There are two Julia constructions that achieve this: start blocks and ; Chains. The value of both composite expression constructions is that of the last subexpression. Here's an example of a starting block: julia> z = beginning x = 1 = 2 x + y end 3Sinth since these are fairly small, simple expressions, they can be easily placed on a single line, which is where the ; chain syntax is useful:julia> z = (x = 1; y = 2; x + y) 3This syntax is especially useful with the one-line definition form introduced in Features. Although it is typical, there is no requirement that starting blocks be multiline or that; chains are single-line: julia> start x = 1; y = 2; x + y end 3 julia> (x = 1; y = 2; x + y) 3Conditional evaluationConditional evaluation allows parts of code to be evaluated or not evaluated, depending on the value of a boolean expression. Here is the anatomy of the if-elseif-else conditional syntax:if x < y= println(x= is= less= than= y)= elseif= x=> y println(x is bigger than y) else println(x is equal to y) endIf the condition expression x < y= is= true,= then= the= corresponding= block= is= evaluated;= otherwise= the= condition= expression= x=> y is evaluated and if it is true, the corresponding block is evaluated; if neither expression is true, the other block is evaluated. Here it is in action: function test(x, y) as x < y= println(x= is= less= than= y)= elseif= x=> y println(x larger than y) otherwise println(x equals y) final test (generic function with 1 method) julia> test(1, 2) x is less than y julia> test(2, 1) x is</TResult> </TResult> </T,TResult> </TResult> </TResult> </TException> </TResult> </TResult> </TResult> then y julia> test(1, 1) x equals yThe elseif and otherwise blocks are optional, and as many elseif blocks as desired can be used. The condition expressions in the if-elseif-else construction are evaluated until the first evaluates where, after which the corresponding block is evaluated, and no further condition expressions or blocks are evaluated.if blocks are leaky, i.e. they do not introduce a local range. This means that new variables defined in the if clauses can be used after the if block, even if they have not been defined previously. So we could have defined the test function above as yolia> function test(x, y) as x < y relationship = less than elseif x == y relationship = equal to other relationship = greater than final print(x is, relationship, y.) final test (generic function with 1 method) julia> test(2, 1) x greater than y.De variable relationship is indicated within the if-block, but used outside. However, when you depend on this behavior, make sure that all possible code paths define a value for the variable. The following change in the above function results in a runtime errorjulia> function test(x,y) as x < y relationship = less than elseif x == y relationship = equal to endprintln(x is, relationship, y.) final test (generic function with 1 method) julia> test(1,2) x is less than y. julia> test(2,1) ERROR: UndefVarError: relationship not defined Stacktrace: [1] test(::Int64, :::Int64) at ./none:7if blocks also return a value that may seem non-intuitive to users from many other languages. This value is simply the return value of the last instruction performed in the branch chosen, sojulia> x = 3 3 julia> as x > 0 positive! otherwise negative... end positive! Note that very short conditional statements (one-liners) are often expressed using Short circuit evaluation in Julia, as described in the following section. Unlike C, MATLAB, Perl, Python and Ruby – but like Java, and a few other stricter, typed languages – it's an error if the value of a conditional expression is anything but true or false:julia> as 1 println(true) end ERROR: TypeError: non-boolean (Int64) used in boolean contextThis error indicates that the conditional of the wrong type: Int64 instead of the required Bool.De so-called ternary operator, ?:, is closely related to the as-elseif-else syntax, but is used when a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It owes its name to the only operator in most languages who has three operands: a ? (b) The expression (a), before the expression, is a conditional expression, and the ternary operation shall evaluate the expression (b), :, if the condition a is true or the expression c, after :, if it is false. Notice that the spaces are around? and : are required: an expression such as a?b:c is not a valid ternary expression (but a new line is after both the ? and the :). The easiest way to understand this behavior is to see an example. In the previous example, the println call is shared by all three branches: the only real choice is which literal string to print. This can be written more concisely with the help of the ternaire operator. To be clear, let's try a two-way version first: julia> x = 1; y = 2; julia> println(x < y ? less than: no less than) less than julia> x = 1; y = 0; julia> println(x < y ? less than : no less than) no less than if the expression x < y is true, the entire ternary operator expression evaluates the string less than and otherwise evaluates it to the string no less than. The original three-way example requires the merpole of multiple applications of the ternary operator:julia> test(x, y) = println(x < y ? x is less than y : x > y ? x is larger than y : x equals y) test (generic function with 1 method) julia> test(1, 2) x is less than y julia> test(2, 1) x greater than y julia> test(1 , 1) x is equal to yTo facketening, the operator associates from right to left. It is significant that if-elseif-else, the expressions before and after : are evaluated only if the condition expression to true or false evaluates, respectively:julia> v(x) = (println(x); x) v (generic function with 1 method) julia> 1 < 2 ? v(yes) : v(nee) yes yes julia> 1 > 2 ? v(yes) : v(nee) no neeShort-Circuit EvaluationShort circuit evaluation is quite similar to evaluation. The behaviour can be found in the most compelling programming languages with the && and || Boolean operators: In a series of boolean expressions connected by these operators, only the minimum number of expressions is evaluated as needed to determine the final boolean value of the entire chain. Explicitly, this means that:In the expression a && b, the subexpression b is evaluated only if one evaluates to true. In the expression a || b, the subexpression b is only evaluated if one evaluates it on false. The reasoning is that a &&b must be false if a is false, regardless of the value of b, and also the value of a || b must be true if a is true, regardless of the value of b. Both && and || right, but && has a higher priority than || Does. It is easy to experiment with this behavior:julia> t(x) = (println(x); true) t (generic function with 1 method) julia> f(x) = (println(x); false) f (generic function with 1 method) julia> t(1) && t(2) 1 1 2 true julia> t(1) && f(2) 1 2 false julia> f(1) && t(2) 1 false julia> f(1) && f(2) 1 false julia gt&; t(1) || t(2) 1 real julia> t(1) || f(2) 1 real julia> f(1) || 1 2 true julia> f(1) || f(2) 1 2 falseU easily experiment with the association and priority of different combinations of && and || Operators. This behaviour is often used in Julia to provide an alternative to very short if Instead of being <cond> <statement>end, one can write <cond>&&&&<statement>(which can be read as: <cond>and then <statement>). In the same way, instead of if ! <cond> <statement>end one can write <cond>|| <statement>(which could be like: or else) <cond> <statement>read. For example, a recursive factorial routine can be defined as follows:julia> fact function(n::Int) n >= 0 || error(n must be non-negative) n == 0 && return 1 n * fact(n-1) end fact (generic function with 1 method) julia> fact(5) 120 julia> fact(0) 1 julia> fact(-1) ERROR: n must be non-negative Stacktrac: [1] error at ./error.jl:33 [inlined] [2] fact(::Int64) at ./none:2 [3] scopeBoolean operations without short circuit evaluation can be performed with the bitwise boolean operators introduced in Mathematical Operations and Elementary Functions: & & &. These are normal functions, which happen to support the syntax of the infix operator, but always evaluate their arguments:julia> f(1) & t(2) 1 2 false julia> t(1) | t(2) 1 2 trueNet as condition expressions used in if, elseif or ternary operator, the operands of & & or || boolean values (true or false). The use of a non-boolean value anywhere except the last entry in a conditional chain is an error:julia> 1 &en true ERROR: TypeError: non-boolean (Int64) used in boolean context On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned, depending on the previous conditionals:julia> true && (x = (1, 2, 3)) (1, 2, 3) julia> false && (x = (1, 2, 3)) falseRepeated evaluation: LoopsThere are two constructs for repeated evaluation of expressions: the while loop and the for loop. Here's an example of a while loop: julia> i = 1; julia> while I <= 5= println(i)= global= i= +=1 end= 1= 2= 3= 3= 4= while= while= while= evaluates= the= condition= expression= (i=></=> &lt== 5= in= this= case),= or= if= if= when= the while= = is= first= reached,= the= the= never= evaluated.= for= loop= makes= common= repeated= evaluation= write.= since= counting= up= and= =loop== while= while= loop= is= is= so= common,= it= can= expressed= more= concisely= with= a= for= loop:julia>for i = 1:5 println(i) end 1 2 3 4 5Here de 1:5 is a range object, which is the order of the numbers 1 1 , 2, 3, 4, 5. The for loop iterates by these values, assigning each one in turn to the i. A fairly important distinction between the previous while loop shape and the for loop shape is the range in which the variable is visible. If the variable i is not entered in another scope, in the for loop form, it is only visible within the for loop, and not outside/after. You'll need a new interactive session instance or other variable name to test this: julia> for j = 1:5 println(j) end 1 2 3 4 5 julia></=> </statement> </cond> </statement> </cond> </statement> </cond> </statement> </cond> </statement> </cond> </statement> </cond> ERROR: UndefVarError: j undefined scope of variables for a detailed explanation of the variable scope and how it works in Julia.In general, the for loop construction can repeat over a container. In these cases, the alternative (but completely equivalent) keyword in or ∈ is usually used instead of =, since it makes the code better read: julia> for i in [1,4,0] println(i) end 1 4 0 julia> for s ∈ [foo,bar,baz] println(s) end foo bar bazVarious types of iterable containers will be introduced and discussed in later sections of the manual (see , for example, multidimensional arrays). It is sometimes useful to end the repetition for a while before the test condition is falsified or stop repeating in a pre-loop before reaching the end of the iterable object. This can be achieved with the pause keyword: julia> i = 1; julia> while true println(i) as I >= 5 break end global i += 1 end 1 2 3 4 5 julia> for j = 1:1000 println(j) as j >= 5 break end end end 1 2 3 4 5Without the keyword break, the above while loop would never end on its own, and the for loop would repeat up to 1000. These loops are both abandoned early using pause. In other circumstances, it is useful to be able to stop an iteration and move immediately to the next one. The keyword remain achieved this: julia> for i = 1:10 if I % 3 != 0 continue to end println(i) end 3 6 9This is a somewhat contrived example, because we can produce the same behavior more clearly by denying the condition and placing the println call in the as block. In realistic use, there is more code to evaluate after the continuation, and often there are several points from which one continues to call. Multiple nested loops can be combined into one outer loop, which constitute the cartesian product of its iterables: julia> for i = 1:2, j = 3:4 println(i, j)) end (1, 3) (1, 4) (2, 3) (2, 4)With this syntax, iterables can still refer to outer loop variables; for example for i = 1:n, j = 1:i is valid. But a pause statement within such a loop leaves the whole nest of loops, not just the inner. Both variables (i and j) are set to their current iteration values each time the inner loop is run. Therefore, commands to i will not be visible for later iterations:julia> for i = 1:2, j = 3:4 println(i, j)) i = 0 end (1, 3) (1, 4) (2, 3) (2, 4)If this example is rewritten to use a keyword for each variable, the output would be different: the second and fourth values would contain 0.Exception Handling If an unexpected condition occurs , a function may not be able to return a reasonable value to

the caller. In cases, it may be best for the exceptional condition to either terminate the program while printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances, then allow that code to take the appropriate action. Built-in exceptionsSuits exceptionsSssions when an unexpected condition has occurred. The built-in exceptions below include all interrupting the normal flow of control. For example, the sqrt function throws a DomainError if it is applied to a negative fair value:julia> sqrt(-1) ERROR: DomainError with -1.0: sqrt only gives a complex result if it is called with a complex argument. Try sqrt (Complex(x)). Stacktrace: [...] You define your own exceptions in the following way: julia> struct MyCustomException <: Exception endThe throw functionExceptions can be created explicit with throw. For example, a function defined only for non-tongative numbers can be written to toss a DomainError if the argument is negative:julia> f(x) = x>=0? exp(-x): throw(DomainError(x, argument must be nonnegative)) f (generic function with 1 method) julia> f(1) 0.36787944117144233 julia> f(-1) ERROR: DomainError with -1: argument must be nonnegative Stacktrace: [1] f(::Int64) at ./none:1Note that DomainError is no exception, but a kind of exception. It must be called upon to obtain an exception object:julia> type of (DomainError(nothing)) <: Exception true julia> type of (DomainError) <: Exception falseAdditionally, some exception types take one or more arguments used for error reporting:julia> throw (UndefVarEr Fout(:x)) ERROR: UndefVarError: x undefinedThis mechanism can be easily implemented based on custom exception types based on how UndefVarError was written:julia> struct MyUndefVarError <: Exception var::Symbol end julia> Base.showerror(io:IO, e:::MyUndefVarError) = printing(io, e.var, undefined)ErrorsThe error function is used to produce an ErrorException that interrupts the normal control flow. Let's say we want to stop the execution immediately if the square root of a negative number is taken. To do this, we can define a picky version of the sqrt function that raises an error if the argument is negative: julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error(negative x not allowed) fussy_sqrt (generic function with 1 method) julia> fussy_sqrt(2) 1.4142135623730951 julia> fussy_sqrt(-1) ERROR: negative x not allowed Stacktrace: [412135623730951 julia> fussy_sqrt(-1) ERROR: Negative x Not Allowed Stacktrace: [2) 1] error at ./error.jl:33 [inlined] [2] fussy_sqrt(::Int64) at ./none:1 [3] scopeIf fussy_sqrt is called with a negative value from another function, Instead of trying to continue running the call function, it returns immediately and displays the error message in the interactive session:julia> function println(vóór fussy_sqrt) r = fussy_sqrt(x) println(na fussy_sqrt) retour r einde verbose_fussy_sqrt generieke functie met 1 methode) julia> verbose_fussy_sqrt(2) vóór fussy_sqrt na fussy_sqrt 1,4142135623730951 julia> verbose_fussy_sqrt(-1) vóór fussy_sqrt FOUT : negatief x niet toegestaan Stacktrace: [1] fout bij ./error.jl:33 [inlined] [2] fussy_sqrt bij ./none:1 [inlined] [3] verbose_fussy_sqrt(::Int64) bij ./none:3 ./none:3 scope at the highest levelThe try/catch statementThe try/catch statement ensures that Exceptions are tested and for the graceful handling of things that might normally break your application. For example, in the code below, the square root function would normally make an exception. By placing a try/catch block around it we can soften that here. You choose how you want to process this exception, whether you want to register it, return a placeholder value or as in the case below where we have just printed out an instruction. One thing to think about when deciding how to deal with unexpected situations is that using a try/catch block is much slower than using conditional branching to handle those situations. Below are more examples of using exceptions with a try/catch block:julia> try sqrt (ten) catch e println (You should have entered a numeric value) end You should have entered a numerical valuetry/catch instructions so that the exception can also be stored in a variable. In the following contrived example, the square root of the second element x is calculated if x is indexable, otherwise, it is assumed that x is a real number and the square root:julia> sqrt_second(x) = try sqrt(x[2]) catch y if isa(y, DomainError) sqrt (complex(x[2], 0)) elseif isa(y, BoundsError) sqrt(x) end sqrt_second (generic function with 1 method julia) > sqrt_second([1 4]) 2.0 julia> sqrt_second([1-4]) 0.0 + 2.0im julia> sqrt_second(9) 3.3 0 julia> sqrt_second(-9) ERROR: DomainError with -9.0: sqrt only gives a complex result if it is invoked with a complex argument. Try sqrt (Complex(x)). Stacktrace: [...] Please note that the next catch symbol will always be interpreted as a name for the exception, so care is needed when writing try/catch expressions on a single line. The following code does not work to return the value of x in the event of an error:try bath() catch x endInstead, use a semicolon, or insert a line break after catch:try bad() catch; x end try bad () catch x endThe power of the try/catch construct lies in the ability to immediately divert a deeply nested calculation to a much higher level in the pile of call functions. There are situations where no error has occurred, but the ability to divest the stack and pass a value to a higher level is desirable. Julia provides the rethrow, backtrace, catch_backtrace and Base.catch_stack features for more advanced error handling.finally ClausesIn code that makes status changes or uses resources such as files, there is usually cleanup work closing files) that must be done when the code is complete. Exceptions may complicate this task, as they can cause a code block to shut down before reaching the normal end. The final keyword provides a way to run a particular code when a particular block of code is shut down, regardless of how it closes. For example, here's how we can guarantee that an open file is closed:f= open(file) try # working on file file finally, close(f) endWhen the check leaves the try block (for example, due to a return, or simply ending normally), close(f) is performed. If the block is closed due to an exception, the exception continues to propagate. A catch block can be combined with try and eventually as well. In this case, the final block runs after catch has processed the error. Tasks (also known as Coroutines) Tasks are a control flow function that allows calculations to be suspended and resumed in a flexible manner. We call them here only for completeness; for a full discussion see Asynchronous Programming. Programming.