Scala Advanced Concepts

cerenode

cerenode.io

Generic Class

- Generic classes are classes which take a type as a parameter. They are particularly useful for collection classes.
- Generic classes take a type as a parameter within square brackets `[]`
- One convention is to use the letter `A` as type parameter identifier, though any parameter name may be used.

```
class Stack[A] {
  private var elements: List[A] = Nil
  def push(x: A) { elements = x :: elements }
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

Variances

- Variance is the correlation of subtyping relationships of complex types and the subtyping relationships of their component types.
- Scala supports variance annotations of type parameters of generic classes.

```
class Foo[+A] // A covariant class
class Bar[-A] // A contravariant class
class Baz[A]  // An invariant class
```

Covariance

- A type parameter A of a generic class can be made covariant by using the annotation +A.

```
trait Iterator[+A] {
 def hasNext(): Boolean
 def next(): A
}
```

- The Iterator trait returns the type A and it’s covariant. That means that we can convert Iterator[A] to Iterator[B] whenever B is a base class of A.

Contravariance

- A type parameter A of a generic class can be made contravariant by using the annotation -A

```
trait Writer[-A] {
 def write(a: Value): Unit
}
```

- The Writer trait receives the type A and it’s contravariant. That means that we can convert Writer[A] to Writer[B] whenever B is a derived class of A.

Type Bound

- In Scala, type parameters and abstract types may be constrained by a type bound.
- An *upper type bound* `T <: A` declares that type variable `T` refers to a subtype of type `A`
- While upper type bounds limit a type to a subtype of another type, *lower type bounds* declare a type to be a supertype of another type. The term `B >: A` expresses that the type parameter `B` or the abstract type `B` refer to a supertype of type `A`

Implicits

- An **implicit parameter** is one that can be automatically inferred based on its type and the values in scope, without you needing to pass in the value of the argument explicitly
- An **implicit conversion function** converts one type to another automatically on-demand
- An **implicit class** is a class marked with the implicit keyword. This keyword makes the class's primary constructor available for implicit conversions when the class is in scope.

Ordered

- Extends the Java Comparable interface
- Ordered trait defines some additional concrete methods that depend on the compare method
- Classes that implement this trait can be sorted with scala.util.Sorting and can be compared with standard comparison operators

```
trait Ordered[T] extends Comparable[T] {
    abstract def compare(that: T): Int
    def <(that: T): Boolean = (this compare that) <  0
    def <=(that: T): Boolean = // …
    def >(that: T): Boolean = // …
    def >=(that: T): Boolean = // …
    def compareTo(that: T): Int = compare(that)
}
```

Ordering

- Ordering is a trait whose instances each represent a strategy for sorting instances of a type
- Extends the Java Comparator interface
- An `Ordering[T]` is implemented by specifying `compare(a:T, b:T)`, which decides how to order two instances a and b
- Ordered provide an implicits Ordering

Exception handling

Option

- A special type of exception that can occur is the absence of some value
- Another use case is in parsing input data (user input, JSON, XML, etc.).Instead of throwing an exception for invalid input you simply return a *None* to indicate parsing failed.

Exception handling

Either

- If you need to provide some more information about the failure we can use **Either**
- It has 2 implementations, *Left* and *Right*. Both can wrap a custom type, respectively type L and type R. By convention *Right* is right, so it contains the successful result and *Left* contains the error.

Exception handling

Try

- *Try[T]* is similar to *Either*. It also has 2 cases, *Success[T]* for the successful case and *Failure[T]*for the failure case. The main difference is that the failure can only be of type *Throwable*.

Exception handling

try catch

- In Scala you rarely see usage of try catch and they are typically only used as a last resort.
- Throwing an exception will break your functional composition and probably result in unexpected behaviour for the caller of your function.

Exception handling

Summary

- *Option[T]*, use it when a value can be absent or some validation can fail and you don't care about the exact cause. Typically in data retrieval and validation logic.
- *Either[L,R]*, similar use case as Option but when you do need to provide some information about the error.
- *Try[T]*, use when something Exceptional can happen that you cannot handle in the function. This, in general, excludes validation logic and data retrieval failures but can be used to report unexpected failures.
- *Exceptions*, use only as a last resort. When catching exceptions use the facility methods Scala provides and never *catch { _ => }*, instead use *catch { NonFatal(_) => }*