Type Parameterization

Objectives

- Scala type parameterization
- Information hiding
- Generic classes and Traits
- Variance

Part 1 :: FUNCTIONAL QUEUES

A functional queue is a data structure with three operations: * head = {returns the first element of the queue} * tail = {returns a queue without its first element} * enqueue = {returns a new queue with a given element appended at the end}

Functional queue does not change its contents when an element is appended. Instead, a new queue is returned that contains the element.

Today we will have a practive with effective queue implementation according such certain specification:

```
scala> val q = Queue(1, 2, 3)
q: Queue[Int] = Queue(1, 2, 3)
scala> val q1 = q enqueue 4
q1: Queue[Int] = Queue(1, 2, 3, 4)
scala> q
res0: Queue[Int] = Queue(1, 2, 3)
```

What happens if queue according such specification is mutable?

Purely functional queues also have some similarity with lists. * Both are so called fully persistent data structures * Both support head and tail operations

The difference is that list is usually extended at the front, using a :: operation, a queue is extended at the end, using enqueue.

The problem is in an efficient implementation. It should not be much worse than mutable (imperative) one.

One simple approach to implement a functional queue would be to use a list as **representation type**.

```
class SlowAppendQueue[T] (elems: List[T]) {
    def head = elems.head
    def teil = new SlowAppendQueue(elems.tail)
```

```
    def enqueue(x: T) = new SlowAppendQueue(elems ::: List(x))
}
```

Where is the puddle?

Enqueue operation takes time proportional to the number of elements stored in the queue.

If we want constant time append, we...

... we can try to reverse the order of the elements in the representation list, so that the last element that's appended comes first in the list.

```
class SlowHeadQueue[T](smele: List[T]) {
    def head = smele.last
    def tail = new SlowHeadQueue(smele.init)
    def enqueue(x: T) = new SlowHeadQueue(x :: smele)
}
```

Now enqueue is constant time, but head and tail are not.

The idea is to represent a queue by two lists, called **leading** and **trailing**.

The leading list contains elements towards the front, whereas the trailing list contains elements towards the back of the queue in reversed order. The contents of the whole queue are at each instant equal to

```
leading ::: trailing.reverse
```

To append an element, you just cons it to the trailing list using the :: operator, so enqueueis constant time. Initially empty queue is constructed from successive enqueue operations, the trailing list will grow whereas the leading list will stay empty. Before first **head** or **tail** operation we

use mirroring = {whole trailing list will copied into leading list reversed order}

```
class Queue[T](
    private val leading: List[T],
    private val trailing: List[T]
) {
    private def mirror =
        if (leading.isEmpty)
            new Queue(trailing.reverse, Nil)
        else
        this

    def head = mirror.leading.head

    def tail = {
        val q = mirror
        new Queue(q.leading.tail, q.trailing)
    }

    def enqueue(x: T) = new Queue(leading, x :: trailing)
}
```

What is the complexity of this implementation of queues?

- The mirror operation might take time proportional to the number of queue elements, but only if list leading is empty.
- head and tail call mirror, their complexity might be linear in the size of the queue, too. But the longer the queue gets, the less often mirror is called.

Hometask: Prove that the functional queue is asymptotically efficient and its efficiency is comparable to the efficiency of the imperative queue. Hint: Think in terms of damped computational complexity.

Part 2 :: INFORMATION HIDING

The efficient implementation of Queue is done. But this efficiency is paid for by exposing a needlessly detailed implementation.

The Queue constructor, which is globally accessible, takes two lists as parameters.

It doesn't look like an intuitive representation of a queue.

Private constructors and factory methods

```
class Queue[T] private (
    private val leading: List[T],
    private val trailing: List[T]
)
```

The constructor of Queue is private: it can be accessed only from within the class itself and its **companion object**.

Companion object brief demo

For each class, you can declare a special companion object. This is an object declared in the same file and with the same name as the class. This object has access to the private fields of the class.

```
class Person(val firstName: String, val middleName: String, val lastName: String)
{}

object Person {
  def apply(firstName: String, lastName: String) = new Person(firstName, "",
lastName)
  def apply(firstName: String) = new Person(firstName, "", "")
}

val p = Person("Tagir", "Magomedov")
```

There is no magic here, it's a call to the apply method on the companion object.

Often, additional class constructors declare this way.

```
class Queue[T] private (
    private val leading: List[T],
    private val trailing: List[T]
)
```

Now let's try some stuff

```
scala> new Queue(List(1, 2), List(3))
<console>:9: error: constructor Queue in class Queue cannot
be accessed in object $iw
    new Queue(List(1, 2), List(3))
```

Now that the primary constructor of class Queue can no longer be called from client code, there needs to be some other way to create new queues. We can add an auxiliary constructor for instance:

```
def this() = this(Nil, Nil)
```

As a refinement, the auxiliary constructor could take a list of initial queue elements:

```
def this(elems: T*) = this(elems.toList, Nil)
```

Note: Recall that T* is the notation for repeated parameters.

Another way is to add a factory method that builds a queue from such a sequence of initial elements.

A neat way to do this is to define an object Queue that has the same name as the class being defined and contains an apply method:

```
object Queue {
    def apply[T](xs: T*) = new Queue[T](xs.toList, Nil)
}
```

By placing this object in the same source file as class Queue, you make the object a **companion object** of the class.

Now the apply method in object Queue can create a new Queue object, even though the constructor of class Queue is private.

Note: the factory method is called apply, clients can create queues with an expression such as Queue(1, 2, 3).

As a result, Queue looks to clients as if it was a globally defined factory method. In reality, Scala has no globally visible methods; every method must be contained in an object or a class.

However, using methods named apply inside global objects, you can support usage patterns that look like invocations of global methods.

An alternative: private classes

Another more radical way is to hide the class itself and only export a trait that reveals the public interface of the class.

```
trait Queue[T] {
    def head: T
    def tail: Queue[T]
    def enqueue(x: T) Queue[T]
}

object Queue {
```

```
    def apply[T](xs: T*): Queue[T] = new QueueInternal[T](xs.toList, Nil)

    private class QueueInternal[T](
        private val leading: List[T],
        private val trailing: List[T]
    ) extends Queue[T] {
        private def mirror =
        if (leading.isEmpty)
            new QueueInternal(trailing.reverse, Nil)
        else
            this

    def head: T = mirror.leading.head

    def tail: QueueInternal[T] = {
        val q = mirror
        new QueueInternal(q.leading.tail, q.trailing)
    }

    def enqueue(x: T) = new QueueInternal(leading, x :: trailing)
    }
}
```

Part 3 :: VARIANCE

Now, Queue is a trait, but not a type. Queue is not a type because it takes a type parameter. And now we cannot:

```
scala> def doesNotCompile(q: Queue) = {}
<console>:8: error: class Queue takes type parameters
def doesNotCompile(q: Queue) = {}
```

Instead, trait Queue enables you to specify parameterized types, such as Queue[String],Queue[Int], or Queue[AnyRef]:

```
scala> def doesCompile(q: Queue[AnyRef]) = {}
doesCompile: (q: Queue[AnyRef])Unit
```

Thus, Queue is a trait and Queue[String] is a type. Queue is also called a **type constructor** because you can construct a type with it by specifying a type parameter. The type constructor Queue "generates" a family of types. You can also say that Queue is a **generic trait**.

The combination of type parameters and subtyping poses some interesting questions.

For example, if S is a subtype of type T, then should Queue[S] be considered a subtype of Queue[T]?

If so, you could say that trait Queue is **covariant** (or "flexible") in its type parameter T.

Since we have just one type parameter, we could say simply that Queues are covariant.

Intuitively, all this seems OK, since a queue of Strings looks like a special case of a queue of AnyRefs.

In Scala, however, generic types have by default **nonvariant** (or "rigid") subtyping.

That means that queues of type Queue with different element types would never be in a subtype relationship.

However, you can demand covariant (flexible) subtyping of queues by changing the first line of the definition of class Queue like this:

```
trait Queue[+T] { ... }
```

Prefixing a formal type parameter with a + indicates that subtyping is covariant (flexible) in that parameter. By adding this single character, you are telling Scala that you wantQueue[String], for example, to be considered a subtype of Queue[AnyRef].

The compiler will check that Queue is defined in a way that this subtyping is sound.

In the other hand we have prefix -, which indicates **contravariant** subtyping:

If Queue were defined like this:

```
trait Queue[-T] { ... }
```

then if T is a subtype of type S, this would imply that Queue[S] is a subtype of Queue[T].

Can you image case when such type of Variance is applicable for some kind of functional Lists/Queues/Vectors?

The + and - symbols you can place next to type parameters are called **variance annotations**.

Checking variance annotations

Let's assume that our queue are covariant.

Then, create a subclass of queues that specializes the element type to Int and overrides the enqueue method:

```
class StrangeIntQueue extends Queue[Int] {
    override def enqueue(x: Int) = {
        println(math.sqrt(x))
        super.enqueue(x)
    }
}
```

Now, lets write a counterexample in two lines:

```
val x: Queue[Any] = new StrangeIntQueue
x.enqueue("abc")
```

What the problem is?

The first of these two lines is valid because StrangeIntQueue is a subclass of Queue[Int] and, assuming covariance of queues, Queue[Int] is a subtype of Queue[Any]. The second line is valid because you can append a String to a Queue[Any]. However, taken together, these two lines have the effect of applying a square root method to a string, which makes no sense.

But it's not just mutable fields that make covariant types unsound. The problem is more general.

It turns out that as soon as a generic parameter type appears as the type of a method parameter, the containing class or trait may not be covariant in that type parameter.

```
class Queue[+T] {
    def enqueue(x: T) = ...
}
```

Running a modified queue class we will get:

```
Queues.scala:11: error: covariant type T occurs in
contravariant position in type T of value x
    def enqueue(x: T) =
```

The thing is a reassignable field

```
var x: T
```

is treated in Scala as a getter method

```
def x: T
```

and a setter method

```
def x_=(y: T)
```

As you can see, the setter method has a parameter of the field's type T. So that type may not be covariant.

Lower bounds

Queue[T] cannot be made covariant in T because T appears as a type of a parameter of the enqueuemethod, and that's a negative position.

But there's a way to get unstuck: we can generalize enqueue by making it polymorphic (i.e., giving the enqueue method itself a type parameter) and using a **lower bound** for its type parameter

```
class Queue[+T] (
    private val leading: List[T],
    private val trailing: List[T] ) {
        def enqueue[U >: T](x: U) =
            new Queue[U](leading, x :: trailing)
}
```

The new definition gives enqueue a type parameter U, and with the syntax, `U >: T` defines T as the lower bound for U. As a result, U is required to be a supertype of T.[1] The parameter toenqueue is now of type U instead of type T, and the return value of the method is now Queue[U]instead of Queue[T].

For example, suppose there is a class Fruit with two subclasses, Apple and Orange. With the new definition of class Queue, it is possible to append an Orange to a Queue[Apple]. The result will be a Queue[Fruit].

Lower bounds with queue covariance, this gives the right kind of flexibility for modeling queues of different element types in a natural way.

This shows that variance annotations and lower bounds play well together. They are a good example of type-driven design, where the types of an interface guide its detailed design and implementation.

This observation is also the main reason that Scala prefers declaration-site variance over use-site variance as it is found in Java's wildcards. With use-site variance, you are on your own designing a class. It will be the clients of the class that need to put in the wildcards, and if they get it wrong, some important instance methods will no longer be applicable.

Upper bounds

Let's remember Merge sort for lists. The way you might want to organize such a sort function is by requiring the type of the list to mix in theOrdered trait.

Mixing Ordered into a class and implementing Ordered's one abstract method, compare, you enable clients to compare instances of that class with <, >, <=, and >=.

```
class Person(
val firstName: String,
val lastName: String) extends Ordered[Person] {
```

```
    def compare(that: Person) = {
        val lastNameComparison =
            lastName.compareToIgnoreCase(that.lastName)
        if (lastNameComparison != 0)
            lastNameComparison
        else
            firstName.compareToIgnoreCase(that.firstName)
    }
override def toString = firstName + " " + lastName
}

def orderedMergeSort[T <: Ordered[T]](xs: List[T]): List[T] = {
    def merge(xs: List[T], ys: List[T]): List[T] =
        (xs, ys) match {
            case (Nil, _) => ys
            case (_, Nil) => xs
            case (x :: xs1, y :: ys1) =>
                if (x < y) x :: merge(xs1, ys)
                else y :: merge(xs, ys1)
    }
val n = xs.length / 2
if (n == 0) xs
else {
    val (ys, zs) = xs splitAt n
    merge(orderedMergeSort(ys), orderedMergeSort(zs))
    }
}
```

With the "T <: Ordered[T]" syntax, you indicate that the type parameter, T, has an upper bound,Ordered[T]. This means that the element type of the list passed to orderedMergeSort must be a subtype of Ordered. Thus, you could pass a List[Person] to orderedMergeSort because Person mixes in Ordered.

Now, although the sort function shown serves as a useful illustration of upper bounds, it isn't actually the most general way in Scala to design a sort function that takes advantage of the Ordered trait. For example, you couldn't use the orderedMergeSort function to sort a list of integers, because class Int is not a subtype of Ordered[Int].

Next time we'll show you how to use implicit parameters and context bounds to achieve a more general solution.