

Lecture 7 – Lazy Evaluation

SWS121: Secure Programming

Jihyeok Park



2024 Spring

- Monads
 - Why Monads?
 - Monad in Scala
 - Monad Laws
 - For Comprehensions
 - Examples
- Custom Monads
 - Tree Monad
 - State Monad

In programming languages, **lazy evaluation** is an evaluation strategy that **delays** the evaluation of an expression until its value is actually **needed**.

In programming languages, **lazy evaluation** is an evaluation strategy that **delays** the evaluation of an expression until its value is actually **needed**.

Scala supports lazy evaluation in different ways:

- Lazy Values
- By-Name Parameters
- Lazy Lists
- Views for Collections

1. Lazy Values (`lazy val`)

Call-By-Need Evaluation

Why Lazy Values?

2. By-Name Parameters

Call-By-Need vs Call-By-Name

Examples

By-Name Parameters with Lazy Values

3. Lazy Lists

Example: Natural Numbers

Example: Even Numbers

Example: Fibonacci Numbers

Example: Prime Numbers

4. Views for Collections

Example: Find Palindromes

1. Lazy Values (`lazy val`)

Call-By-Need Evaluation

Why Lazy Values?

2. By-Name Parameters

Call-By-Need vs Call-By-Name

Examples

By-Name Parameters with Lazy Values

3. Lazy Lists

Example: Natural Numbers

Example: Even Numbers

Example: Fibonacci Numbers

Example: Prime Numbers

4. Views for Collections

Example: Find Palindromes

Consider the following example:

```
val x: Int = {  
    println("Initializing x")  
    42  
}  
println("Before accessing x")  
println(x)  
println("After accessing x")
```

Consider the following example:

```
val x: Int = {  
    println("Initializing x")  
    42  
}  
println("Before accessing x")  
println(x)  
println("After accessing x")
```

The output of the above code is:

```
Initializing x  
Before accessing x  
42  
After accessing x
```


Consider the following example:

```
val x: Int = {  
    println("Initializing x")  
    42  
}  
println("Before accessing x")  
println(x)  
println("After accessing x")
```

The output of the above code is:

```
Initializing x  
Before accessing x  
42  
After accessing x
```

We can delay the initialization of a value by defining it as a **lazy val**.

The basic form of lazy evaluation in Scala is the `lazy val` declaration.

The basic form of lazy evaluation in Scala is the `lazy val` declaration.

```
lazy val x: Int = {  
  println("Initializing x")  
  42  
}  
println("Before accessing x")  
println(x)  
println("After accessing x")
```

The basic form of lazy evaluation in Scala is the `lazy val` declaration.

```
lazy val x: Int = {  
  println("Initializing x")  
  42  
}  
println("Before accessing x")  
println(x)  
println("After accessing x")
```

The output of the above code is:

```
Before accessing x  
Initializing x  
42  
After accessing x
```

```
def f: Int = { println("Evaluating f"); 42 }  
println(f)    // First access  
println(f)    // Second access  
lazy val x: Int = { println("Initializing x"); 42 }  
println(x)    // First access  
println(x)    // Second access
```

Is it same as the function call?

```
def f: Int = { println("Evaluating f"); 42 }  
println(f)    // First access  
println(f)    // Second access  
lazy val x: Int = { println("Initializing x"); 42 }  
println(x)    // First access  
println(x)    // Second access
```

Is it same as the function call? **No!** because the value of a `lazy val` is **cached** after the first evaluation.

The output of the above code is:

```
Evaluating f  
42  
Evaluating f  
42  
Initializing x  
42  
42
```

We call the evaluation strategy of `lazy val` as **call-by-need**.

We call the evaluation strategy of `lazy val` as **call-by-need**.

In call-by-need evaluation, the expression is evaluated **once** and the result is **cached** for future accesses.

We call the evaluation strategy of `lazy val` as **call-by-need**.

In call-by-need evaluation, the expression is evaluated **once** and the result is **cached** for future accesses.

The expression is evaluated **only when needed**.

We call the evaluation strategy of `lazy val` as **call-by-need**.

In call-by-need evaluation, the expression is evaluated **once** and the result is **cached** for future accesses.

The expression is evaluated **only when needed**.

It supports the following properties at the same time:

- ① **Immutability**
- ② **On-Demand Evaluation**
- ③ **Caching for Reuse**

Why Lazy Evaluation?

We can **avoid unnecessary heavy computation**:

```
// Heavy computation
lazy val f: Int = {
  Thread.sleep(5000) // sleep for 5 second
  42
}
val x = scala.io.StdIn.readInt
val y = if (x > 0) f else 0
```

Why Lazy Evaluation?

We can **avoid unnecessary heavy computation**:

```
// Heavy computation
lazy val f: Int = {
  Thread.sleep(5000) // sleep for 5 second
  42
}
val x = scala.io.StdIn.readInt
val y = if (x > 0) f else 0
```

The above code will sleep for 5 seconds only if x is positive.

Why Lazy Evaluation?

We can **avoid unnecessary heavy computation**:

```
// Heavy computation
lazy val f: Int = {
  Thread.sleep(5000) // sleep for 5 second
  42
}
val x = scala.io.StdIn.readInt
val y = if (x > 0) f else 0
```

The above code will sleep for 5 seconds only if x is positive.

It means that the heavy computation is done only when needed.

Why Lazy Evaluation?

We can **change the evaluation order**:

```
lazy val z: Int = y + x
lazy val x: Int = { println("Initializing x"); 1 }
lazy val y: Int = { println("Initializing y"); 2 }
println(z)
```

Why Lazy Evaluation?

We can **change the evaluation order**:

```
lazy val z: Int = y + x
lazy val x: Int = { println("Initializing x"); 1 }
lazy val y: Int = { println("Initializing y"); 2 }
println(z)
```

The output of the above code is:

```
Initializing y
Initializing x
3
```

Why Lazy Evaluation?

We can **change the evaluation order**:

```
lazy val z: Int = y + x
lazy val x: Int = { println("Initializing x"); 1 }
lazy val y: Int = { println("Initializing y"); 2 }
println(z)
```

The output of the above code is:

```
Initializing y
Initializing x
3
```

The evaluation of `z` is done after the evaluation of `y` and `x`.

1. Lazy Values (`lazy val`)

Call-By-Need Evaluation

Why Lazy Values?

2. By-Name Parameters

Call-By-Need vs Call-By-Name

Examples

By-Name Parameters with Lazy Values

3. Lazy Lists

Example: Natural Numbers

Example: Even Numbers

Example: Fibonacci Numbers

Example: Prime Numbers

4. Views for Collections

Example: Find Palindromes

Another way to achieve lazy evaluation in Scala is by using **by-name parameters** with the prefix `=>` in the parameter type.

```
// return y if x is positive, otherwise return 0
def f(x: Int, y: => Int): Int = {
  println(s"Calling f with x = $x")
  val result = if (x > 0) y else 0
  println(s"Returning $result")
  result
}
f(1, { println("Evaluating y"); 42 })
f(-1, { println("Evaluating y"); 42 })
```

Another way to achieve lazy evaluation in Scala is by using **by-name parameters** with the prefix `=>` in the parameter type.

```
// return y if x is positive, otherwise return 0
def f(x: Int, y: => Int): Int = {
  println(s"Calling f with x = $x")
  val result = if (x > 0) y else 0
  println(s"Returning $result")
  result
}
f(1, { println("Evaluating y"); 42 })
f(-1, { println("Evaluating y"); 42 })
```

The output of the above code is:

```
Calling f with x = 1
Evaluating y
Returning 42
Calling f with x = -1
Returning 0
```

```
def f(x: => Int): Int = {  
  println(s"Calling f with x = $x")  
  x + x + x  
  println(s"Returning $result")  
}  
f({ println("Evaluating x"); 42 })
```

Is it also **call-by-need** evaluation?

```
def f(x: => Int): Int = {  
  println(s"Calling f with x = $x")  
  x + x + x  
  println(s"Returning $result")  
}  
f({ println("Evaluating x"); 42 })
```

Is it also **call-by-need** evaluation? **No!** because it evaluates the expression **every time** it is used unlike **lazy val**.

The output of the above code is:

```
Calling f with x = 42  
Evaluating x  
Evaluating x  
Evaluating x  
Returning 126
```

The **call-by-need** evaluation satisfies the following properties:

- ① **Immutability**
- ② **On-Demand Evaluation**
- ③ **Caching for Reuse**

The **call-by-name** evaluation evaluates the following properties:

- ① **Immutability**
- ② **On-Demand Evaluation**
- ③ **No Caching for Reuse** (evaluate every time it is used)

The common use case of call-by-name parameters is **short-circuiting**:

```
def myIf(cond: Boolean, thenExpr: => Int, elseExpr: => Int): Int =  
  if (cond) thenExpr  
  else elseExpr  
def f(x: Int): Int = myIf(  
  x <= 0,  
  { println("then"); -x },  
  { println("else"); x }  
)  
println(f(-3))  
println(f(7))
```

The common use case of call-by-name parameters is **short-circuiting**:

```
def myIf(cond: Boolean, thenExpr: => Int, elseExpr: => Int): Int =  
  if (cond) thenExpr  
  else elseExpr  
def f(x: Int): Int = myIf(  
  x <= 0,  
  { println("then"); -x },  
  { println("else"); x }  
)  
println(f(-3))  
println(f(7))
```

The output of the above code is:

```
then  
3  
else  
7
```


The `getOrElse` method in `Option` is also implemented using call-by-name parameters:

```
def f(opt: Option[Int]): Int = opt.getOrElse {  
    println("Evaluating default value"),  
    42  
}  
println(f(Some(10)))  
println(f(None))
```

The `getOrElse` method in `Option` is also implemented using call-by-name parameters:

```
def f(opt: Option[Int]): Int = opt.getOrElse {  
  println("Evaluating default value"),  
  42  
}  
println(f(Some(10)))  
println(f(None))
```

The output of the above code is:

```
10  
Evaluating default value  
42
```

We can combine **call-by-name parameters** with **lazy val** to achieve the properties of both.

```
def f(x: Int, y: => Int): Int = {  
  println(s"Calling f with x = $x")  
  lazy val z = y  
  val result = if (x > 0) z + z + z else 0  
  println(s"Returning $result")  
}  
f(1, { println("Evaluating y"); 42 })  
f(-1, { println("Evaluating y"); 42 })
```

We can combine **call-by-name parameters** with **lazy val** to achieve the properties of both.

```
def f(x: Int, y: => Int): Int = {  
  println(s"Calling f with x = $x")  
  lazy val z = y  
  val result = if (x > 0) z + z + z else 0  
  println(s"Returning $result")  
}  
f(1, { println("Evaluating y"); 42 })  
f(-1, { println("Evaluating y"); 42 })
```

The output of the above code is:

```
Calling f with x = 1  
Evaluating y  
Returning 126  
Calling f with x = -1  
Returning 0
```

1. Lazy Values (`lazy val`)

Call-By-Need Evaluation

Why Lazy Values?

2. By-Name Parameters

Call-By-Need vs Call-By-Name

Examples

By-Name Parameters with Lazy Values

3. Lazy Lists

Example: Natural Numbers

Example: Even Numbers

Example: Fibonacci Numbers

Example: Prime Numbers

4. Views for Collections

Example: Find Palindromes

In Scala, a `LazyList` is a **lazy** version of a `List` that evaluates elements **only when needed**.

In Scala, a LazyList is a **lazy** version of a List that evaluates elements **only when needed**.

We can create a LazyList using the `#::` operator.

```
def create(n: Int): LazyList[Int] = {  
  println(s"Creating lazy list with $n")  
  n #:: create(n + 1)  
}  
lazy val xs: LazyList[Int] = create(0)  
println(xs(0))  
println(xs(2))
```

In Scala, a LazyList is a **lazy** version of a List that evaluates elements **only when needed**.

We can create a LazyList using the `#::` operator.

```
def create(n: Int): LazyList[Int] = {  
  println(s"Creating lazy list with $n")  
  n #:: create(n + 1)  
}  
lazy val xs: LazyList[Int] = create(0)  
println(xs(0))  
println(xs(2))
```

The output of the above code is:

```
Creating lazy list with 0  
0  
Creating lazy list with 1  
Creating lazy list with 2  
2
```


Example: Natural Numbers

Using the LazyList, we can create **infinite** lists.

Example: Natural Numbers

Using the LazyList, we can create **infinite** lists.

The following code creates a LazyList of all **natural numbers**:

```
nat ---- ...
```

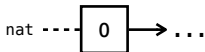
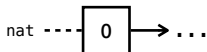
```
nat ---- ...
```

```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
```

Example: Natural Numbers

Using the LazyList, we can create **infinite** lists.

The following code creates a LazyList of all **natural numbers**:

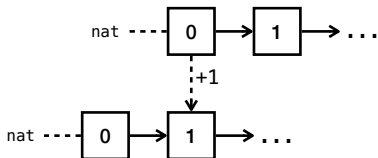


```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
println(nat(0))           // 0
```

Example: Natural Numbers

Using the LazyList, we can create **infinite** lists.

The following code creates a LazyList of all **natural numbers**:

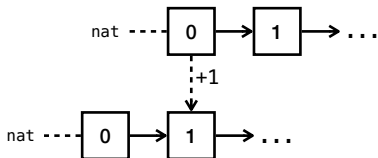


```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
println(nat(0))           // 0
println(nat(1))           // 1
```

Example: Natural Numbers

Using the LazyList, we can create **infinite** lists.

The following code creates a LazyList of all **natural numbers**:

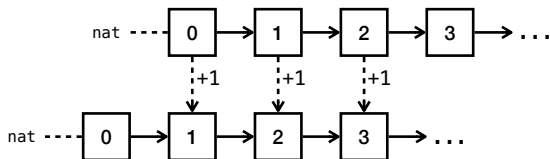


```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
println(nat(0))           // 0
println(nat(1))           // 1
println(nat(0))           // 0
```

Example: Natural Numbers

Using the LazyList, we can create **infinite** lists.

The following code creates a LazyList of all **natural numbers**:

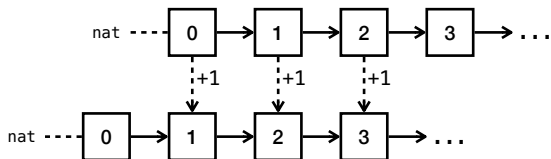


```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
println(nat(0))           // 0
println(nat(1))           // 1
println(nat(0))           // 0
println(nat(3))           // 3
```

Example: Natural Numbers

Using the LazyList, we can create **infinite** lists.

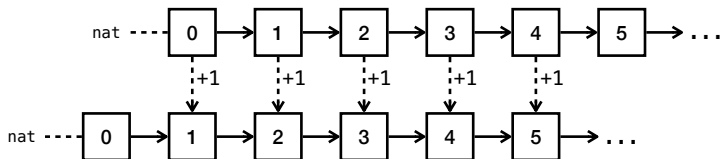
The following code creates a LazyList of all **natural numbers**:



```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
println(nat(0))           // 0
println(nat(1))           // 1
println(nat(0))           // 0
println(nat(3))           // 3
println(nat(2))           // 2
```

Using the LazyList, we can create **infinite** lists.

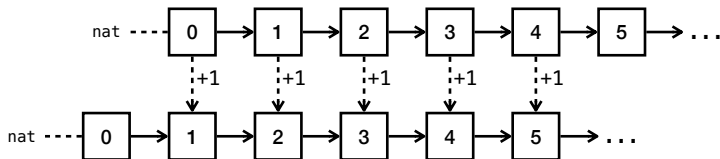
The following code creates a LazyList of all **natural numbers**:



```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
println(nat(0))      // 0
println(nat(1))      // 1
println(nat(0))      // 0
println(nat(3))      // 3
println(nat(2))      // 2
println(nat(5))      // 5
```


Using the LazyList, we can create **infinite** lists.

The following code creates a LazyList of all **natural numbers**:



```
lazy val nat: LazyList[Int] = 0 #:: nat.map(_ + 1)
println(nat(0))           // 0
println(nat(1))           // 1
println(nat(0))           // 0
println(nat(3))           // 3
println(nat(2))           // 2
println(nat(5))           // 5
println(nat(4))           // 4
```

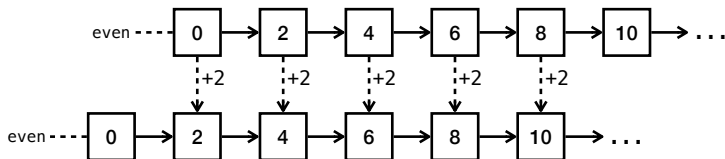
Example: Even Numbers

We can create other infinite lists using `LazyList`.

Example: Even Numbers

We can create other infinite lists using LazyList.

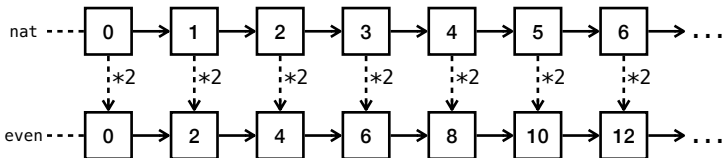
For example, we can create a LazyList of all **even numbers**:



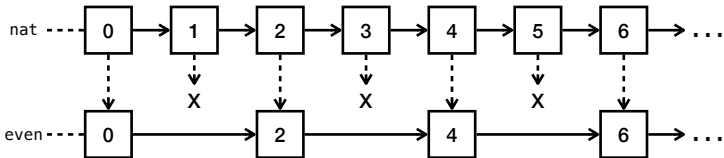
```
lazy val even: LazyList[Int] = 0 #:: even.map(_ + 2)
```

Example: Even Numbers

We can utilize the `map` or `filter` methods to create even numbers.



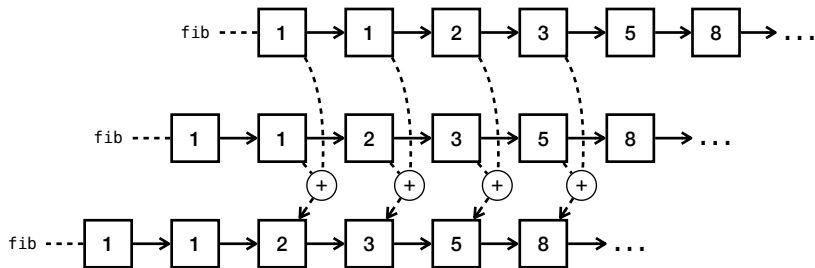
```
lazy val even: LazyList[Int] = nat.map(_ * 2)
```



```
lazy val even: LazyList[Int] = nat.filter(_ % 2 == 0)
```

Example: Fibonacci Numbers

We can utilize the zip method to create the **Fibonacci numbers**.



```
lazy val fib: LazyList[Int] = 1 #:: 1 #:: (fib zip fib.tail).map(_ + _)
```

where `fib.tail` takes all elements except the first element.

Example: Prime Numbers

We can create the natural numbers starting from a specific number using `LazyList.from` method:

```
lazy val nats: LazyList[Int] = LazyList.from(3)
// 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
```

Example: Prime Numbers

We can create the natural numbers starting from a specific number using `LazyList.from` method:

```
lazy val nats: LazyList[Int] = LazyList.from(3)
// 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
```

Besides the Fibonacci numbers, we can also create the **prime numbers**:

```
lazy val primes: LazyList[Int] = 2 #:: LazyList
    .from(3)
    .filter {
        i => primes
            .takeWhile(p => p * p <= i)
            .forall(p => i % p != 0)
    }
```

where the `takeWhile` method returns elements until the condition is satisfied, and the `forall` method checks if all elements satisfy the condition.

1. Lazy Values (`lazy val`)

Call-By-Need Evaluation

Why Lazy Values?

2. By-Name Parameters

Call-By-Need vs Call-By-Name

Examples

By-Name Parameters with Lazy Values

3. Lazy Lists

Example: Natural Numbers

Example: Even Numbers

Example: Fibonacci Numbers

Example: Prime Numbers

4. Views for Collections

Example: Find Palindromes


```
def inc(x: Int): Int = {  
  println(s"inc: $x -> ${x + 1}")  
  x + 1  
}  
  
def double(x: Int): Int = {  
  println(s"double: $x -> ${x * 2}")  
  x * 2  
}  
  
(1 to 100).map(inc).map(double)(5)
```

```
def inc(x: Int): Int = {  
  println(s"inc: $x -> ${x + 1}")  
  x + 1  
}  
  
def double(x: Int): Int = {  
  println(s"double: $x -> ${x * 2}")  
  x * 2  
}  
  
(1 to 100).map(inc).map(double)(5)
```

The output of the above code is:

```
inc: 1 -> 2  
inc: 2 -> 3  
...  
double: 2 -> 4  
double: 3 -> 6  
...  
14
```

The view method creates a **view** of the collection that **computes elements on demand**.

```
def inc(x: Int): Int = {  
  println(s"inc: $x -> ${x + 1}")  
  x + 1  
}  
def double(x: Int): Int = {  
  println(s"double: $x -> ${x * 2}")  
  x * 2  
}  
(1 to 100).view.map(inc).map(double)(5)
```

The output of the above code is:

```
inc: 6 -> 7  
double: 7 -> 14  
14
```

The view method creates a **view** of the collection that **computes elements on demand**.

```
def inc(x: Int): Int = {  
  println(s"inc: $x -> ${x + 1}")  
  x + 1  
}  
  
def double(x: Int): Int = {  
  println(s"double: $x -> ${x * 2}")  
  x * 2  
}  
  
(1 to 100).view.map(inc).map(double)(5)
```

The output of the above code is:

```
inc: 6 -> 7  
double: 7 -> 14  
14
```

We can optimize the computation using view method.

```
val xs: IndexedSeqView[Int] = (1 to 100).view
```

```
val xs: IndexedSeqView[Int] = (1 to 100).view
```

The view type supports the same operations as the original collection (e.g., `IndexedSeqView` supports similar operations as `IndexedSeq`).

```
val ys = xs.view.map(_ + 1).filter(_ % 2 == 0).take(5)
```

```
val xs: IndexedSeqView[Int] = (1 to 100).view
```

The view type supports the same operations as the original collection (e.g., `IndexedSeqView` supports similar operations as `IndexedSeq`).

```
val ys = xs.view.map(_ + 1).filter(_ % 2 == 0).take(5)
```

We can force to perform the computation using the `to` method or `to*` method (e.g., `toVector`, etc.).

```
ys.to(Vector) // Vector(2, 4, 6, 8, 10)  
ys.toVector   // Vector(2, 4, 6, 8, 10)
```

```
val xs: IndexedSeqView[Int] = (1 to 100).view
```

The view type supports the same operations as the original collection (e.g., IndexedSeqView supports similar operations as IndexedSeq).

```
val ys = xs.view.map(_ + 1).filter(_ % 2 == 0).take(5)
```

We can force to perform the computation using the to method or to* method (e.g., toVector, etc.).

```
ys.to(Vector) // Vector(2, 4, 6, 8, 10)  
ys.toVector   // Vector(2, 4, 6, 8, 10)
```

It computes elements **in need** but **not caches** the results. (**call-by-name**).

```
val zs = xs.view.map(x => { println(x); x + 1 })  
zs(42) // prints 42  
zs(42) // prints 42
```


Consider the following huge list of words:

```
val words: List[String] = // a huge list of words
```

Consider the following huge list of words:

```
val words: List[String] = // a huge list of words
```

Let's implement a code to find the first palindrome word in the first 10,000 words, or return `None` if no palindrome word is found.

Consider the following huge list of words:

```
val words: List[String] = // a huge list of words
```

Let's implement a code to find the first palindrome word in the first 10,000 words, or return `None` if no palindrome word is found.

The possible basic implementation is:

```
def isPalin(s: String): Boolean = s == s.reverse  
def findPalin(words: List[String]): Option[String] = words.find(isPalin)  
findPalin(words.take(10_000))
```

However, it always requires to create a new list of the first 10,000 words.

Consider the following huge list of words:

```
val words: List[String] = // a huge list of words
```

Let's implement a code to find the first palindrome word in the first 10,000 words, or return None if no palindrome word is found.

The possible basic implementation is:

```
def isPalin(s: String): Boolean = s == s.reverse  
def findPalin(words: List[String]): Option[String] = words.find(isPalin)  
findPalin(words.take(10_000))
```

However, it always requires to create a new list of the first 10,000 words.

To avoid creating a new list, we can use the view method.

```
findPalin(words.view.take(10_000))
```

Consider the following huge list of words:

```
val words: List[String] = // a huge list of words
```

Let's implement a code to find the first palindrome word in the first 10,000 words, or return None if no palindrome word is found.

The possible basic implementation is:

```
def isPalin(s: String): Boolean = s == s.reverse  
def findPalin(words: List[String]): Option[String] = words.find(isPalin)  
findPalin(words.take(10_000))
```

However, it always requires to create a new list of the first 10,000 words.

To avoid creating a new list, we can use the view method.

```
findPalin(words.view.take(10_000))
```

The view method helps to **optimize** the computation in diverse ways.

1. Lazy Values (`lazy val`)

Call-By-Need Evaluation

Why Lazy Values?

2. By-Name Parameters

Call-By-Need vs Call-By-Name

Examples

By-Name Parameters with Lazy Values

3. Lazy Lists

Example: Natural Numbers

Example: Even Numbers

Example: Fibonacci Numbers

Example: Prime Numbers

4. Views for Collections

Example: Find Palindromes

- Generics

Jihyeok Park

`jihyeok_park@korea.ac.kr`

`https://plrg.korea.ac.kr`