Laziness and Streams

About me

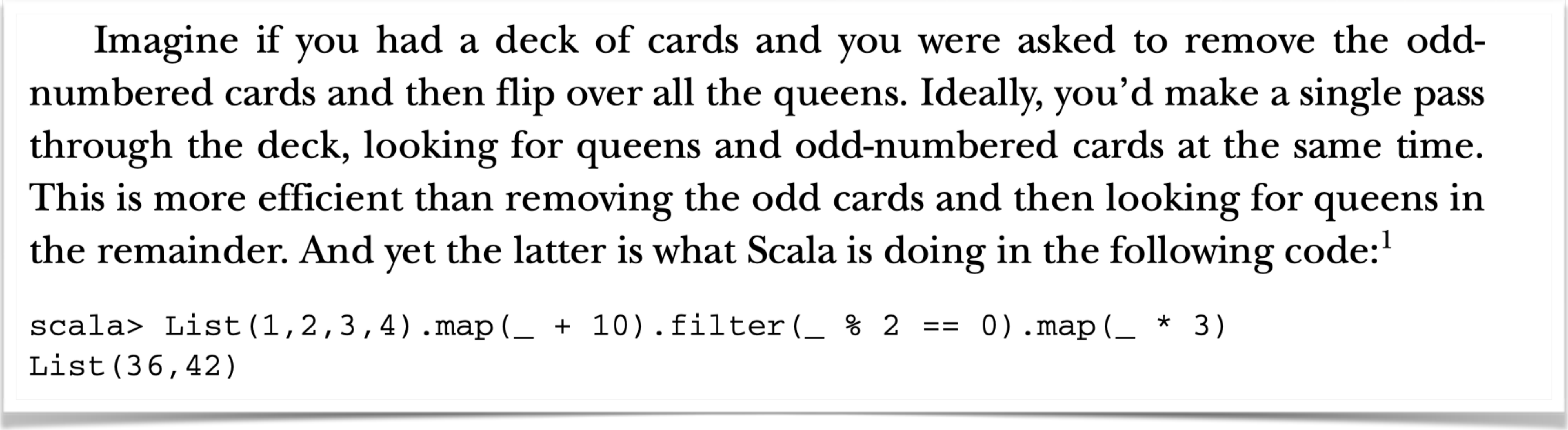
- Assistant prof in CS Department. Office: 4D08
- Programming Language Theory and Implementation
- Graduated from Ecole Polytechnique, France

Today's class

- Concepts around **laziness**
- Implementation of a **lazy** list — Stream
- Applications

“Laziness is the first step towards efficiency.”

Functional programming languages need laziness.



Imagine if you had a deck of cards and you were asked to remove the odd-numbered cards and then flip over all the queens. Ideally, you'd make a single pass through the deck, looking for queens and odd-numbered cards at the same time. This is more efficient than removing the odd cards and then looking for queens in the remainder. And yet the latter is what Scala is doing in the following code:[1]

```
scala> List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)
List(36,42)
```

Chiusano et al. p64

Laziness in Scala

```
val x = {println (“eager”); 5}
```

Earger

```
lazy val y = {println (“lazy”); 4}
```

Lazy

DEMO

Discuss in groups of two/three people (3 min)

```
val x = {println (“eager”); 5}
```

Earger

```
lazy val y = {println (“lazy”); 4}
```

Lazy

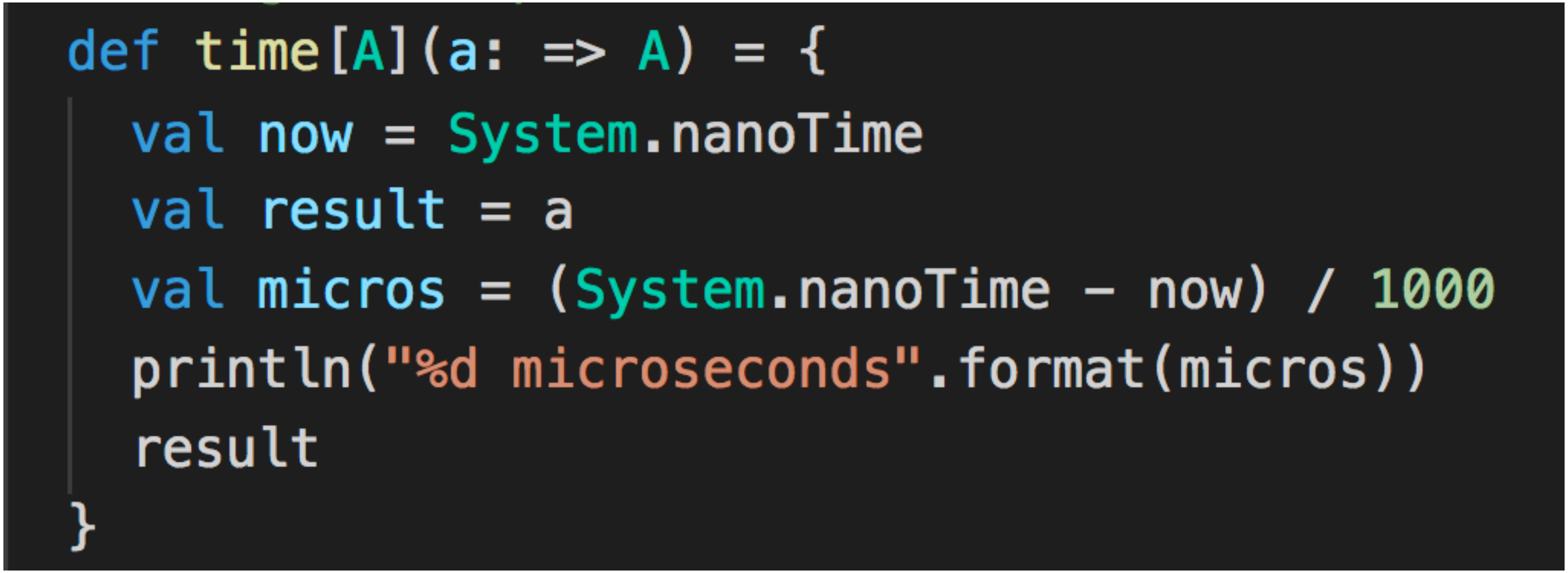
What would be x+x+y+y ?

Laziness in function calls

- A *lazy function* passes arguments without evaluating them. Example: &&, | |
- A *strict function* does the opposite.

DEMO

Example: lazy function



```scala
def time[A](a: => A) = {
  val now = System.nanoTime
  val result = a
  val micros = (System.nanoTime - now) / 1000
  println("%d microseconds".format(micros))
  result
}
```

DEMO

Some other terms to know

```
val x = {println(“hello"); 42}
```

By-value

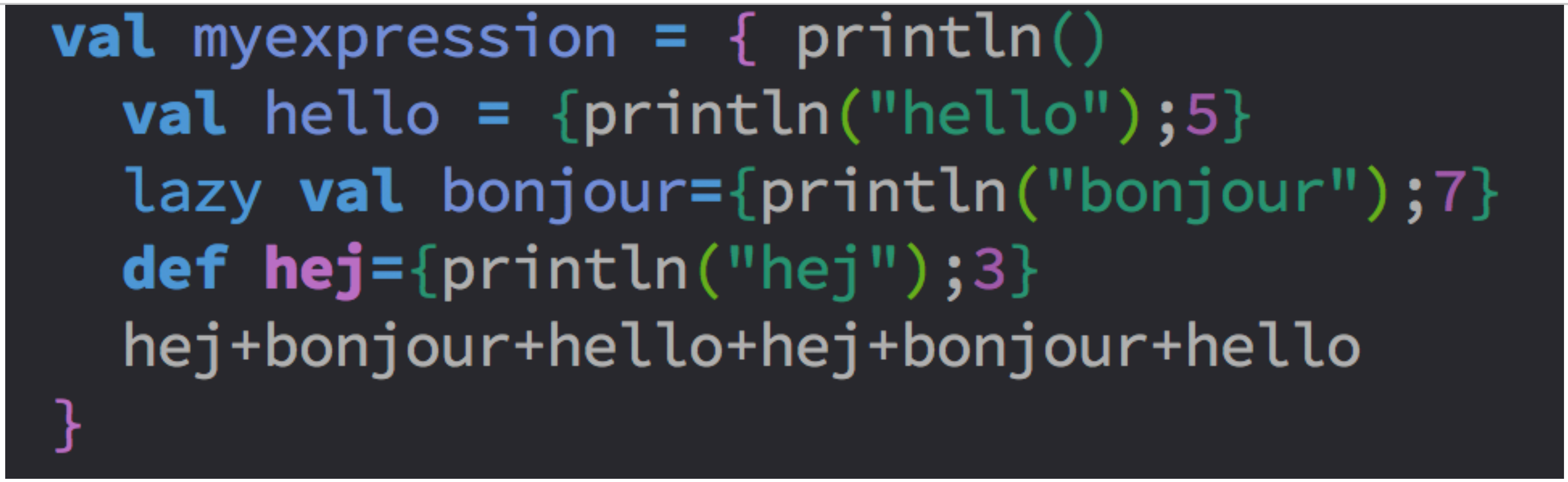
```
def x = {println(“hello"); 42}
```

By-name

```
lazy val x = {println(“hello"); 42}
```

By-need

Discuss in groups of 2-3 people (6 min)



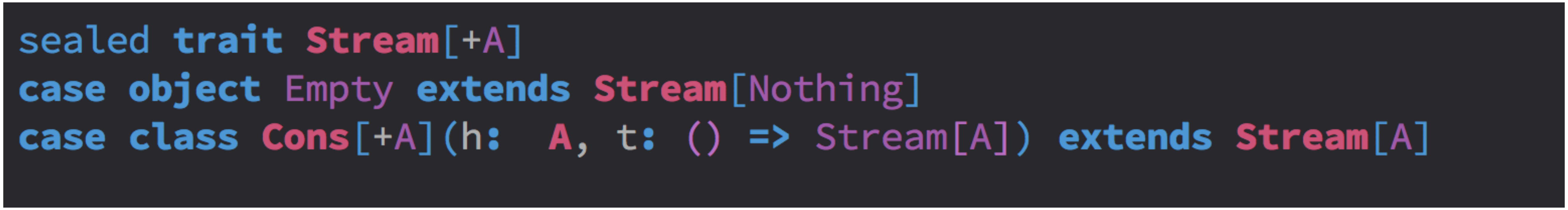
```
val myexpression = { println()
  val hello = {println("hello");5}
  lazy val bonjour={println("bonjour");7}
  def hej={println("hej");3}
  hej+bonjour+hello+hej+bonjour+hello
}
```

- What would be the outputs?

DEMO

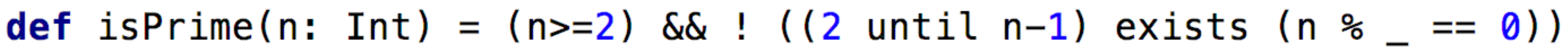
Streams

Stream = “List with a lazy tail”



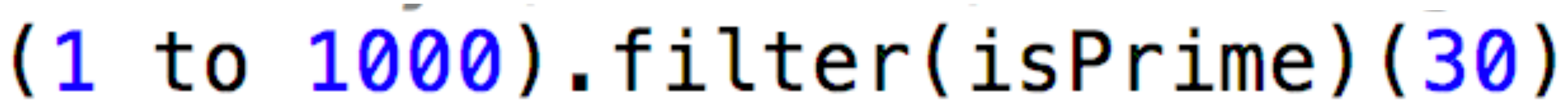
```
sealed trait Stream[+A]
case object Empty extends Stream[Nothing]
case class Cons[+A](h:  A, t: () => Stream[A]) extends Stream[A]
```

- Stream[A] is either Empty, or Cons (h, t)
    - where h is of type A, and t is a Stream
- +A for covariance: Stream[A]<Stream[B] if A<B
- The tail is not to be evaluated (being lazy)

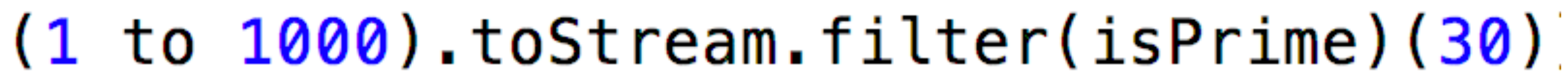


```
def isPrime(n: Int) = (n>=2) && ! ((2 until n-1) exists (n % _ == 0))
```

A textbook example:
Find the 31-st prime number



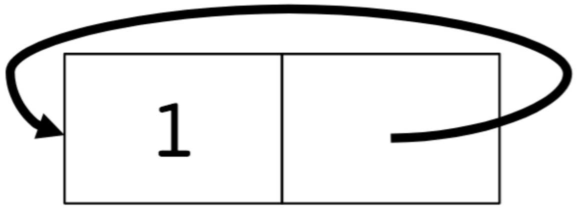
```
(1 to 1000).filter(isPrime)(30)
```



```
(1 to 1000).toStream.filter(isPrime)(30)
```

DEMO

Another Example: Infinite List



- val ones: Stream[Int] = Stream.cons(1, ones)
- ones(1000)

DEMO

Conclusions for today’s class

- Laziness brings efficiency
- Implementation via different evaluation strategies
- Put things together => Stream