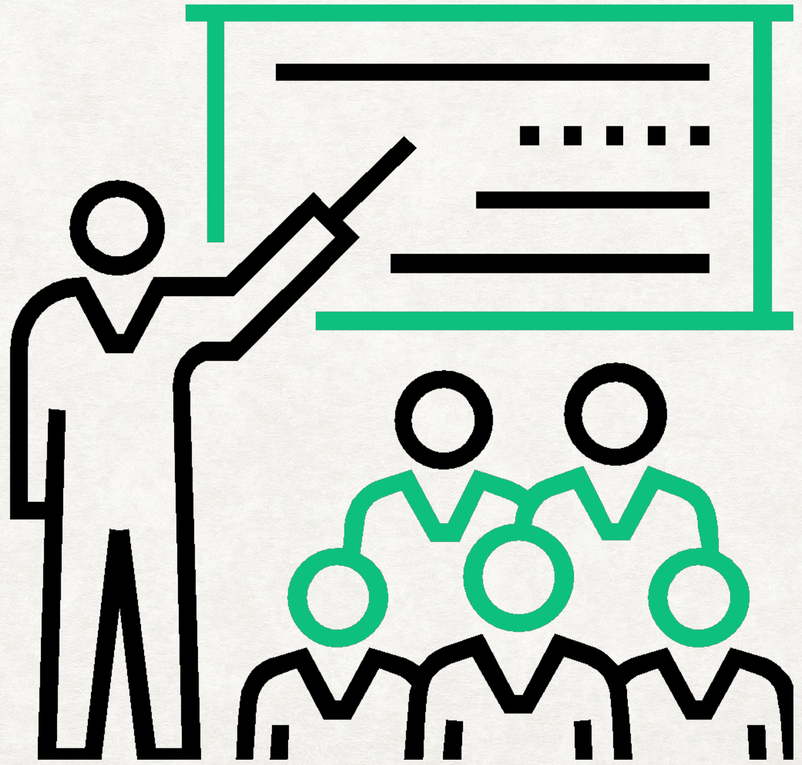
ADVANCED FUNCTIONAL THINKING

DR SURBHI SARASWAT
SCHOOL OF COMPUTER SCIENCE

SCALA LAZY SEQUENCES

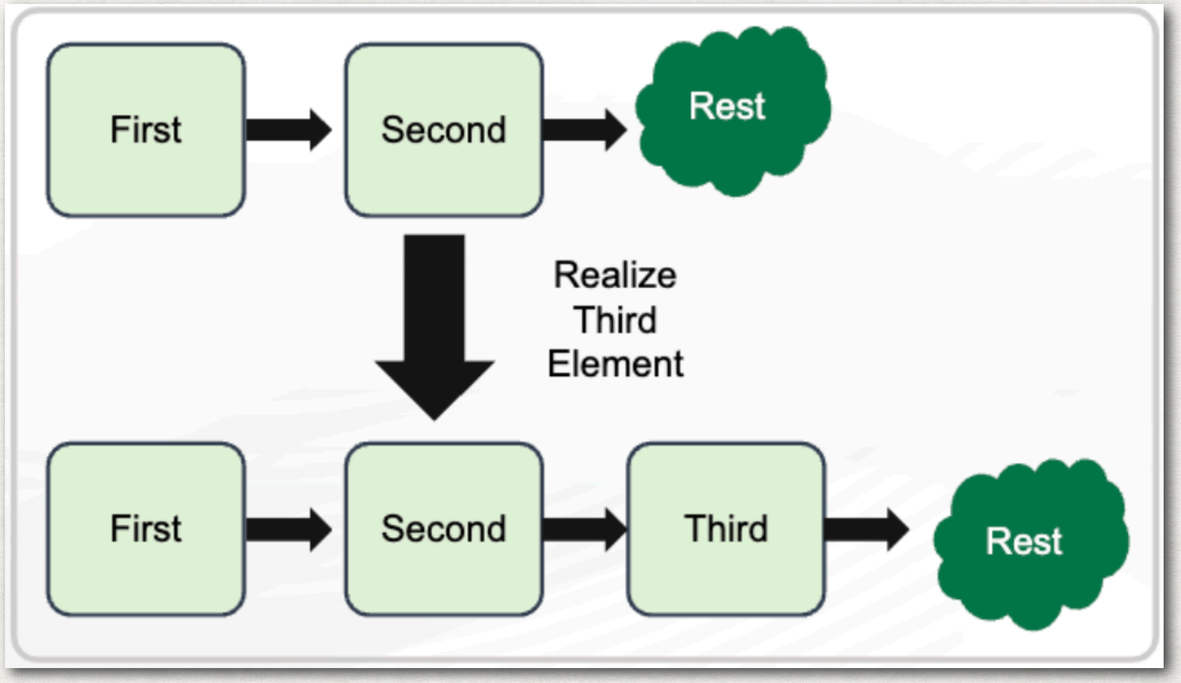
- Lazy Sequences
- Proxy Patterns
- Scala Streams
    - Finite
    - Infinite
    - Recursive



LAZY SEQUENCES

It is used to create a sequence whose members are computed only when needed. This allows us to easily stream results from a computation and to work with infinitely long sequences.

- Each sequence of element can be dealt one at a time.
- Generally, before processing, it is not required to release the entire sequence.
- When we create an element, we call that realising the element. Once realised, the elements are put into a case using Memoization, which means we only need to realise each element in the sequence once.



LAZY SEQUENCES

Example 1: Create an infinite sequence of all integers

```
val integers = Stream.from(0) //create a sequence of all integers.
      val someints = integers take 5 // take the first five integers
      someints foreach println //Prints first five integers

Output:
0
1
2
3
4
```

Scala has built-in support for Lazy Sequence in its Stream library. Perhaps the simplest thing we can do with a lazy sequence is to create an infinite sequence of all integers. Scala's Stream library has a method that does just that, called from. According to the ScalaDoc, it will "create an infinite stream starting at start and incrementing by step."

LAZY SEQUENCES

Example 2: Create an infinitely long sequence of pseudorandom numbers

```
val generate = new Random() //create random number generator.
      val randoms = Stream.continually(generate.nextInt)  // pass the
method
      val aFewRandoms = randoms take 5 //take 5 random numbers
       aFewRandoms foreach println  //Prints five random numbers

Output:
326862669
-473217479
-1619928859
785666088
1642217833
```

Let's take a look at a slightly fancier instance of lazy sequence that uses another method in Scala's Sequence library. The continually method creates an infinitely long sequence by repeatedly evaluating the expression passed into here.

Let's use this to create an infinitely long sequence of pseudorandom numbers. To do so, we create a new random number generator in the val generate, and then we pass generate. nextInt in the continually method, as illustrated in the above code.

SCALA STREAM (FINITE)

A Stream is a similar data structure to a list except that the elements of the **Stream** will be lazily computer. As a result, you can have infinitely long **Streams**!

Streams: A stream is generally a lazy and linear sequence (or list) collection.

Scala Streams:

```
scala> import scala.Stream
import scala.Stream

scala> var ms = Stream("a","b","c","d","e")
ms: scala.collection.immutable.Stream[String] = Stream(a, ?)

scala> print(ms)
Stream(a, ?)
```

We have created a stream of finite length using Stream(). But why does print(ms) shows us Stream(a, ?) ?

It is clear that the print function is just showing the first element, and for the rest of the elements, it is showing ?. The character ? is telling us that only the first element of the stream has been evaluated. And the remaining elements will be evaluated only when required.

Indexing the elements of a stream:

Elements of a stream can be indexed using ().

For example, if I want to get the first element of the list, the following code snippet will do it:

scala> ms(0)

res14: String = a

We have got the first element of the stream and have understood that indexing will be started from 0 not from 1 as some programming language support:

scala> ms(2)

res15: String = c

scala> print(ms)

Stream(a, b, ?)

The ms(2) have printed c and stream has evaluated its value up to b which means up to the second element.

SCALA STREAM (INFINITE)

Streams in Scala are constructed with #::

```
scala> val str = 1 #:: 2 #:: 3 #:: Stream.empty
str: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

Lists are constructed with the :: operator, streams are constructed with the similar-looking #::. Here is a simple example of a stream containing the integers 1, 2, and 3:

scala> val str = 1 #:: 2 #:: 3 #:: Stream.empty

str: scala.collection.immutable.Stream[Int] = Stream(1, ?)

In this example the head of the stream is 1, and the tail of it has 2 and 3. If you notice the tail is not printed, though, because it hasn't been computed yet! The streams are computed lazily, and the toString method of a stream is careful not to force any extra evaluation.

SCALA STREAM (INFINITE RECURSIVE)

Fibonacci Sequence:

```
scala> def fibFrom(a: Int, b: Int): Stream[Int] = a #:: fibFrom(b, a + b)
fibFrom: (a: Int,b: Int)Stream[Int]

scala> val fibs = fibFrom(1, 1).take(7)
fibs: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> fibs.toList
res9: List[Int] = List(1, 1, 2, 3, 5, 8, 11)
```

This function is deceptively simple. The first element in the sequence is exactly a, and the rest of the sequence is the Fibonacci sequence starting with b followed by a + b. The most trickiest part is computing this sequence without causing an infinite recursion. If the function is used :: instead of #::, then every call to the function will result in another call, thus causing an infinite recursion. Because it is using #::, though, the right-hand side is not evaluated until it is requested.

METHODS OF THE STREAM CLASS

Append:

```
scala> var strm = Stream(1,2,3,4,5)
strm: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> var strm1 =  Stream(6,7,8)
strm1: scala.collection.immutable.Stream[Int] = Stream(6, ?)

scala> var astrm = strm.append(strm1)
astrm: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

Length:

```
scala> astrm.length
res1: Int = 8
```

METHODS OF THE STREAM CLASS

Mathematical Functions:

```
scala> astrm.sum
res5: Int = 36

scala> astrm.max
res2: Int = 8

scala> astrm.min
res3: Int = 1
```

Count:

```
scala> var strm = Stream(1,2,3,4,5)
scala> strm.count(x => x%2==0)
res6: Int = 2
```

Find:

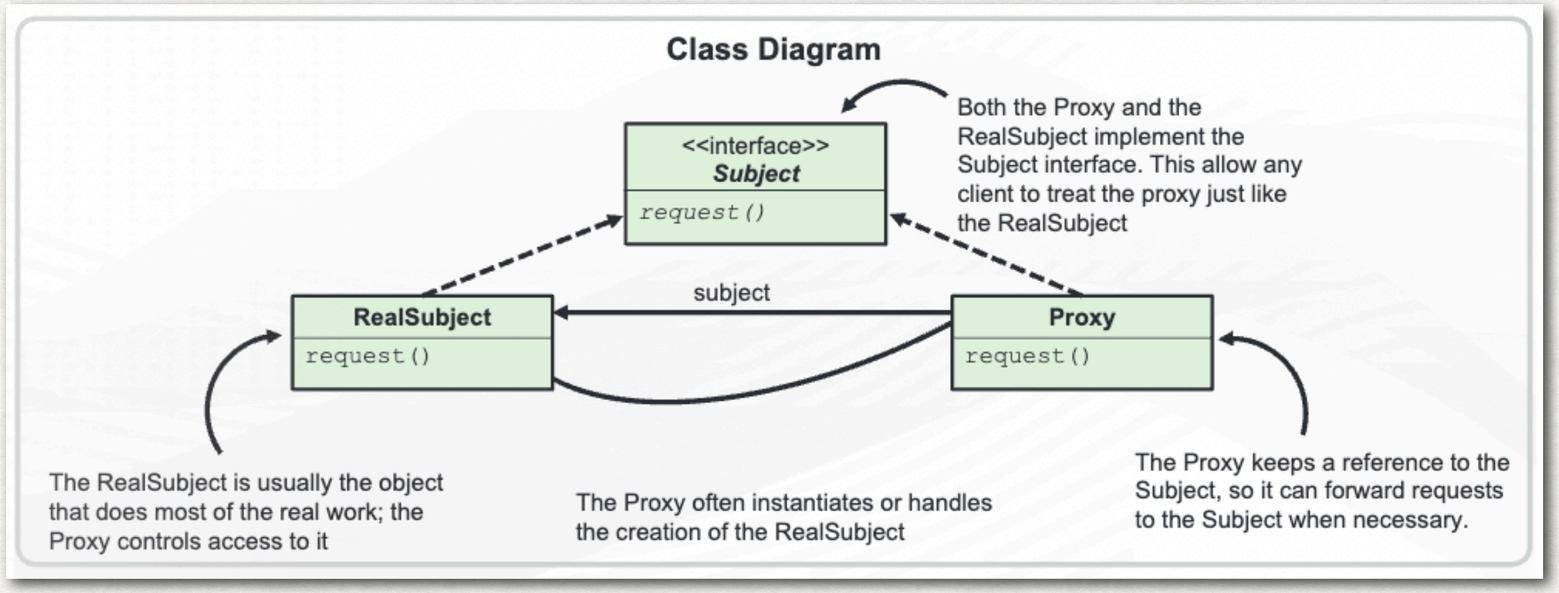
```
scala> strm
res7: scala.collection.immutable.Stream[Int] = Stream(1, 2, 3, 4, 5)

scala> strm.find(x => x >3)
res10: Option[Int] = Some(4)
```

there are two types of options: some() and none().
none is used to handle none vlaues for examples if theer is no evaluation result is produced then it is used.

security
reduce latency
traffic control

THE PROXY PATTERN

- A proxy is a class functioning as an interface to something else.
- Proxy Pattern is a structural design which is used to create a representative object that controls to another object, which may be remote, expensive to create, or in need of securing.

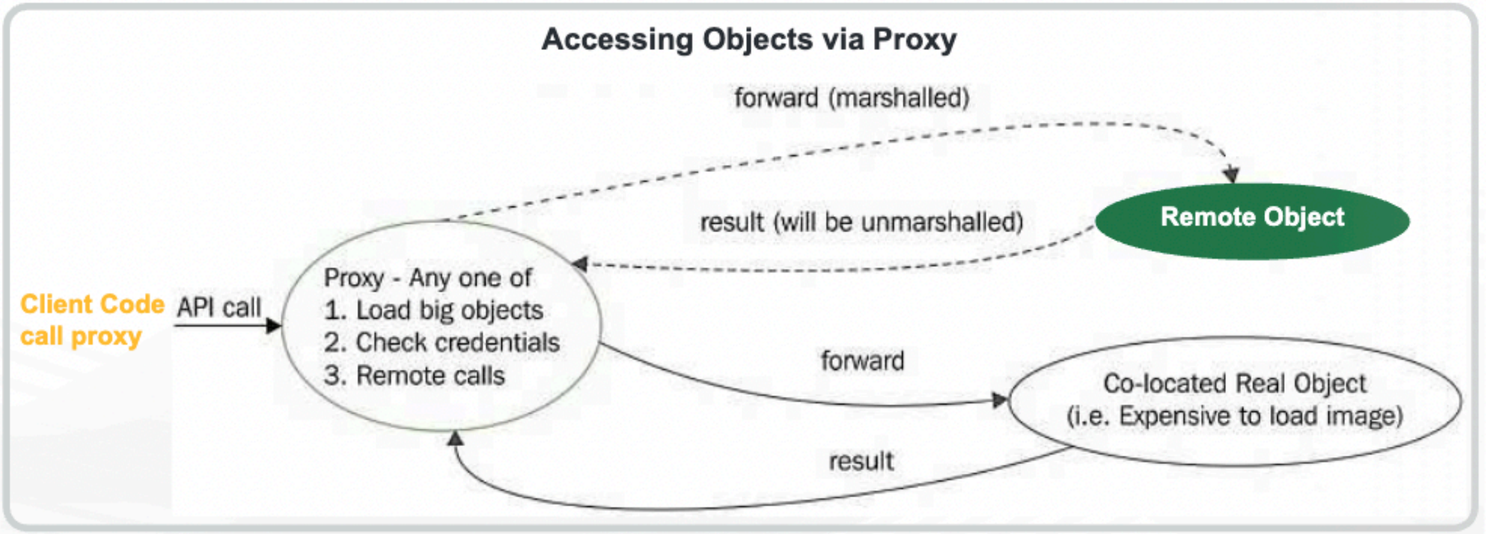


THE PROXY PATTERN

- First we have a Subject, which provides an interface for the RealSubject and the Proxy.
- By implementing the same interface, the proxy can be substituted for the RealSubject anywhere it occurs.
- The RealSubject is the object that does the real work. It's the object that the proxy represents. The proxy holds a reference to the RealSubject.
- In some cases, the Proxy may be responsible for creating and destroying the RealSubject. Clients interact with the RealSubject through the Proxy.
- The proxy also controls the access to the RealSubject;
  - this control may be needed if the Subject is running on a remote machine,
  - if the Subject is expensive to create, or
  - if access to the subject needs to be protected in some way.

THE PROXY PATTERN: EXAMPLE

- Let us say we have to access the metric data from a database.
  - Assume this is a static data and we do not want to hit the database each time we need this data in our application.
  - Also, we are using an external API to access this data from the database and the default behaviour is to hit the db every time the data is requested.
- Considering we have a static data, we want to build something which will hit the db only the first time we request the data and cache it, so that we can return the data from cached data instead of hitting the db each time.
- We can build a proxy to solve this.
  - First, we have a QueryCommand, which provides an interface for RealSubject and Proxy.
  - We have a real-object MetricTableCommand & a proxy MetricTableCommandProxy.
  - The MetricTableCommandProxy holds a reference to the MetricTableCommand.



The salient points of the above image are as follows:

- Client code calls proxy class.
- The actual image class holds the actual image. This image would take a lot of time to get loaded into memory. We want to load it on demand.
- The proxy class, proxy to the actual image.
- We cache the path to the actual image.
- First time, we load the thumbnail image.
- The next time someone asks us to render it again (maybe by clicking it again), we load the actual image and render it.