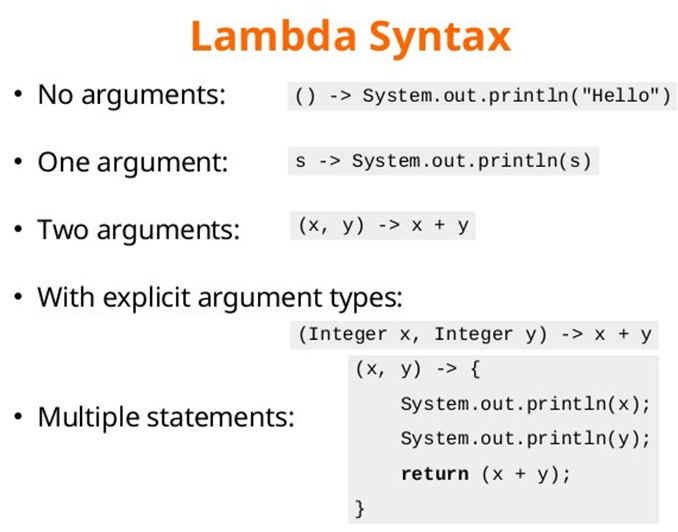
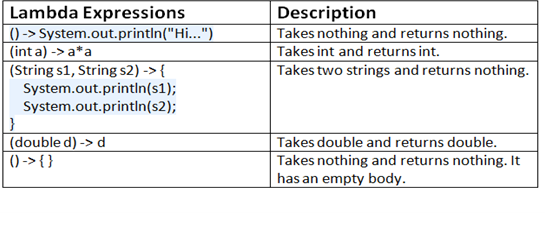
**Lambda Expressions**

* A lambda expression is a short block of code which **takes in parameters and performs some logic**.
  + Lambda expressions are like methods, but they go unnamed, and they can be implemented right in the body of a method. This allows you to create **single use, unnamed (anonymous) methods**, that can take parameters.
* **The most basic syntax of a lambda expression is** parameter(s) -> expression





* Under the hood, lambdas are instances of **Functional Interfaces,** which are Interfaces that have exactly one abstract method. When you write a lambda, you are providing an implementation for that single abstract method.
  + **Every lambda is an instance of some Functional Interface**. There are plenty of built-in Functional Interfaces, and you can make custom ones as well. They’re commonly used in Java, with Streams and Threads, for instance. (Runnable?)
  + **Why use lambdas over just defining a method?**
    - The main use case I see is for shorthand and single use method functionality. If you don’t want to make a whole concrete method, lambdas are a good choice.
    - Lambdas can also be passed in as parameters for a method. If I want a method to be able to use some undefined functionality, we could use lambdas as an argument.
    - Also, lots of built-in Java methods use lambdas so we’re stuck using them in some cases.

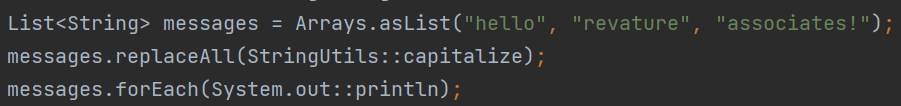
**Streams**

* Streams allow us to **process a collection of elements**. They do not store elements - rather they take in a group of elements (from a file, data structure, etc.) and **perform some operations on it**.
  + We can then take that stream-processed data and store/use it elsewhere in our application.
  + It’s important to note that **Streams do not change the original data**, rather they output some transformation of the data.
* Streams consist of **Intermediate Operations, which perform modifications** on the dataset and **Terminal Operations, which complete (or *terminate*) the modifications** and may or may not return a value.
  + **Intermediate Operations include:**
    - **map()** - applies a given function to every element of the stream
    - **filter()** - takes out any elements that don’t match a given criteria
    - **sorted()** - sorts the elements by their natural order (alphabetically, alphanumerically, etc.)
  + **Terminal Operations include:**
    - **collect()** - returns the stream as a given object (List? Set? Array?)
    - **toList()** - returns the stream as a list
    - **forEach()** - performs a given function to every element found in the stream (then terminates the stream, unlike map())
    - **average()** - returns the average of the values
    - **reduce()** - reduces the elements of a stream to a single value
      * We can define a functionality for this (like summing all the numbers)

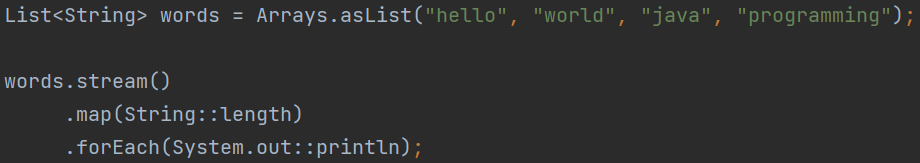
\*Saw Streams in HelloThreadsStreams\*

**Method Reference Syntax**

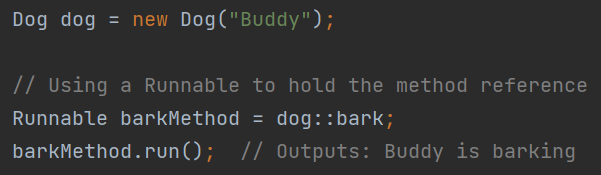
* Method references are a way to **refer to a method without directly executing it.**
  + They're used to create **more readable lambda expressions** by referring to existing methods, and it’s also just quicker and can be more convenient.
  + Method references use the **:: operator**
* There are 4 types of method references we can use:  
  + **Static methods** 
    - Syntax - ClassName::staticMethodName

  
 Using two different static methods, one from StringUtils and one from System.out

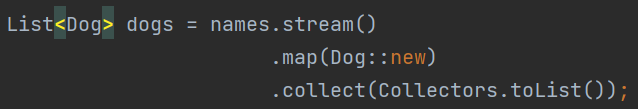
* + **Instance methods of an arbitrary object of a particular type**
    - Syntax - ClassName::instanceMethodName

  
In this example, we use the reference on any object of a specific Type (String), so the number of objects or the specific object we’re on doesn’t matter.

* + **Instance methods of** **particular objects**
    - Syntax - instance::instanceMethodName

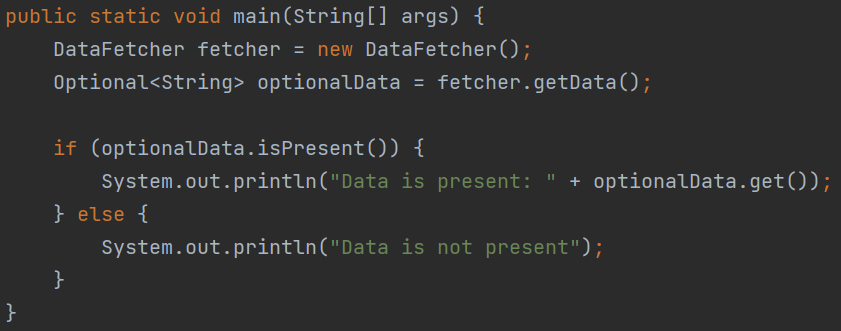
 Assume we have a Dog class with a bark() method. We can assign it to a Thread in a shorthand way using method reference syntax. This one isn’t as useful as the others in my opinion.

* + **Constructor reference**
    - Syntax - ClassName::new

  
 This is a fancy example - we make a List of Dog objects based on a List of Dog names.

**Optional Class**

* The Optional class in Java is a container object that **may or may not contain a non-null value**. It was introduced to help developers **deal with null values and avoid NullPointerExceptions.**
  + The main idea behind Optional is to provide a solution for representing optional values instead of using null. It's an expressive, safe way to control the output of a method that may not always return a value.
* Optional has two convenient methods that help developers determine what to do with a method’s returned data (or lack thereof):
  + **isPresent():** This method returns true if there is a value present in the Optional object, otherwise it returns false. It's often used in a conditional to check if a value exists before trying to use it.
  + **isEmpty():** This method is the opposite of isPresent(). It returns true if the Optional object is empty (i.e., it contains null), otherwise it returns false.

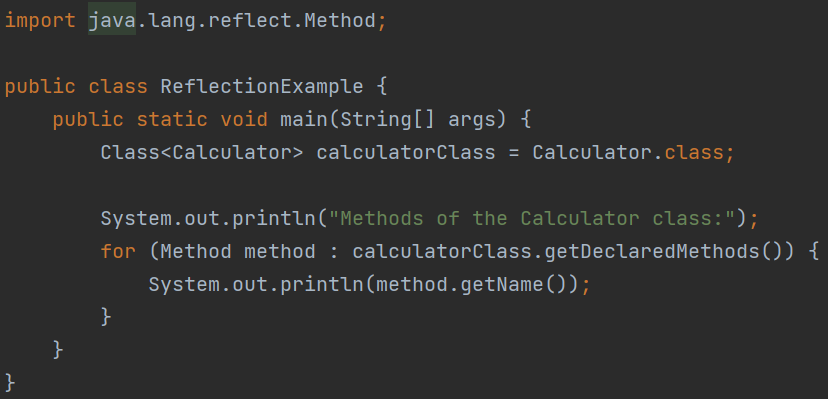
  
 Here’s an example of Optional in the wild. We see this often when we use Spring Data, since some of the methods from JpaRepository return Optionals.

* Lastly, notice the .**get() method, which is how we extract data from an Optional.**
  + If we don’t use .get(), we can’t get the desired data out of the Optional container.

**Reflection API**

* The Reflection API is a powerful feature that allows you to **inspect and manipulate classes, interfaces, constructors, methods, and fields at runtime.** Pretty much any Java entity. It's part of the java.lang.reflect package and can be used to:
  + **Inspect a class's structure:** You can retrieve information about a class's methods, fields, constructors, and superclasses.
  + **Create an instance of a class:** You can create new instances of a class without knowing its name at compile time. (Spring uses this for beans)
  + **Inspect and modify the runtime behavior of applications:** You can change field values and invoke methods on objects and classes.

Here’s a simple example using java.lang.reflect.Method



Just be careful not to overuse reflection, as it can lead to very vague, nonspecific code.

**Some data structure topics in the curriculum**

**(maybe won’t have time to deeply cover these together)**

**Sets**

* Like a set in math - an unordered collection of unique objects.
  + Unique means no duplicate values
* No particular order, and no index. Thus, we can’t iterate through them without using .iterator() (read below)
* One Null element max, since there are no duplicates
* Methods
  + **.iterator ->** a lambda that returns an Iterator with the same elements as the Set.
    - ITERATORS HAVE INDEXES which is why one may choose to do this.
      * We can use the following methods to *iterate* through an Iterator:
        + **.next():** Advances through the Iterator by one index and returns the value it passed.

This lets us go through all the Iterator’s values

* + - * + **.hasNext():** Returns true if there is an element to the right of the current index.

This lets us see if there are any remaining values to iterate over.

* + - * + **Iterator does not have a get() style method like you would find in a List.**
  + **.add():** add an element to the Set
  + **.remove():** remove an element from the Set
  + **.size():** gets the length of the Set (as an int)
  + **.contains()** - returns a boolean of true if the object given is contained within the Set
* Implementations (Concrete Classes that implement the Set Interface)
  + **HashSet:** Typical general use Set. All the same attributes as stated above.
  + **TreeSet:** Ordered set, lets you get elements in their natural order
    - Natural order as in alphabetical/numeric

**Queues**

* FIFO: First in, First out
* Queues are like waiting in a line (Lines are called queues in a lot of countries).
  + Thus, queues are ordered, but you can only touch elements on one end
  + Most queues don’t let you insert nulls, but some (like LinkedLists) do.
* Methods
  + **.peek():** looks at first element
  + **.poll():** get/remove first element
  + **.add()**
* Implementations
  + **ArrayDeque** – orders queue elements based on time of insertion
    - ArrayDeques are double ended – elements can be added or removed from the beginning or end of the Queue.
  + **PriorityQueue** - orders queue elements based on natural ordering (alph, num)
    - PriorityQueues are NOT double ended – elements can only be added at the end or removed from the beginning of the Queue.

**Maps (not actually an Iterable - not technically a Collection)**

* Unique Key/Value pairs
  + Keys can’t be null
  + Keys must refer to a value
* NOT iterable like the others
* NOT part of the collections framework
* Lookup/Delete of data is super fast thanks to the key/value structure
* Methods
  + **.put (key k, value v): add elements**
  + **.remove (key k, value v):** remove element
  + **.clear():** remove all elements
  + **.get(key)**: gets the value of a certain key
  + **.containsKey()**
  + **.containsValue()**
  + **.keyset()** -> returns a Set of the keys
  + **.values()** returns a Collection of the values
* Implementations
  + **HashTable** - no order to keys, no nulls
  + **HashMap** - no order to keys, up to one null key, any null values
* Don’t forget to look at the diagram below! And take note of the **Iterable Interface**, which is the parent of all collections (as mentioned in the curriculum with no supporting notes)

