U6S3M3 - Lambda Expressions

named method - a **method with a name** that can **invoked from anywhere in the code** accept parameters and return a value.

named method are used when you want to call the method multiple times from different place in your code.

Named method that takes 2 parameters and returns their sum:

```
public int sum(int n1, int n2) {
    return n1 + n2;
}
```

Named method that takes no parameters and returns a 0:

```
public int giveMeAZero() {
    return 0;
}
```

Lambda expression - a **method without a name** that can **only be invoked where it is defined** it can accept parameters and return a value (aka anonymous methods/functions)

Lambda expression is used where a method is needed in only one placed in the code. (Usually as parameters to other methods)

instead of *function* name you code **->** after the parameters and before the **{**

Lambda expression that takes 2 parameters and returns their sum:

```
(n1, n2) -> {
    return n1 + n2;
}

(n1, n2) -> { return n1 + n2; }
```

Lambda Expression that takes no parameters and returns a 0:

```
() -> {
    return 0;
}

() -> { return 0; }
```

Lambda expression allows the use of a method without creating a class method.

Lambda expressions are frequently used as arguments for other methods.

Some methods require method be passed to them that they then execute as part of thier processing. The methods that are passed are referred to a **callback method/functions**.

When a method is used as a parameter to another method, we refer to the parameter method as a **callback method**.

The method receiving the **callback method**, calls the callback method as part of it's processing.

Callback methods may either named or Lambda expression.

Java Stream Interface methods

The **Stream** interface is an API that may be used to process collections of objects.

A stream is a sequence of objects that supports various methods which can be chained/pipelined to produce the desired result.

The **stream()** method is used to invoke the interface operation methods.

Stream Interface Operation Methods:

Note: Some Stream methods return a Stream object which may need to be converted to the Collections class type needed using the .collect(Collectors.*toxxxxx()*) methods.
For example:

```
.collect(Collector.toList())
.collect(Collectors.toSet())
.collect(Collectors.toMap())
```

forEach() - Stream interface version of Java for-each. It runs a passed in Lambda expression for every element of an array.

In the Lambda expression passed to **forEach**, process the element passed to it.

In following example:

1. Each element in **anArray** is passed to an Lambda expression.
2. The Lambda expression assigns the name **anElement** to the element passed.
3. The Lambda expression processes **anElement**

```
anArray.forEach((anElement) ->  {
                                    System.out.println(anElement)
                                })
```

map() - Using an implied forEach, **return a Stream object** using the return value of the Lambda expression method as the values in the Stream object.

This is **NOT** the same as a Collections class Map<key-type, value-key>

Use use **map()** when you need an new Collections object from processed elements of a Collections object.

In the following example:

1. Each element in **anArray** is passed to an Lambda expression.
2. The Lambda expression assigns the name **anElem** to the element passed.
3. The Lambda expression processes **anElem** and returns a value **(newElem)**
4. **map()** will add the value returned from the Lambda expression to a new Stream object
5. When all elements in **anArray** are processed, **map()** returns the new Stream object
6. Convert the Stream object returned by **map()** to a List object

```
List<data-type> newArray = anArray.map( (anElem) -> {
                                          newElem = ...;
                                          return newElem;
                                        }).collect(Collectors.toList());
```

filter() - Using an implied forEach loop, filter will take an Lambda expression, run each element through it and return a Stream object.

If the **expression returns true**, the **current element will be kept** in the new Stream object.

If the **expression returns false**, the **current element will be dropped** from the new Stream object.

Use **filter()** when you need to create a Collections object from a Collections object based on a condition.

In the following example *(find values in a List that are divisible by 3)*:

1. Each element in **numsToFilter** is passed to an Lambda expression.
2. The Lambda expression assigns the name **aNum** to the element passed.
3. The Lambda expression processes **aNum:**
    a. Use **aNum** in some logic and returns a **true** or **false**
4. **filter()** will add the value to a new Stream object if Lambda expression returned true,
5. When all elements in **numsToFilter** are processed, **filter()** returns the new Stream object.
6. Convert Stream object to a List

```
numsToFilter.stream().filter((aNum) -> {
                                    return (aNum % 3 == 0 ? true : false);
                                    })
                    .collect(Collectors.toList());
```

reduce() - Using an implied forEach loop, **reduce()** will **collapse a Collections object down to one single value** (sometimes referred to as the *reducer*) using the logic of the Lambda expression.

```
reduce(initial-reducer-value, Lambda-expression)
```

Lambda expression receives two parameters:

- the reducer
- the next element to process in the expression

Use **reduce()** when you need to convert a Collections object to a single value.
ex. sum of the values.

In the following example:

1. Each element in **numbersToSum** is passed to an Lambda expression along with a variable to hold the results of all previous calls to the Lambda expression.
2. The Lambda expression assigns the name **sum** to the reducer and **aNumber** to the element passed. The reducer (**sum**) will hold the current result of all calls to the Lambda expression.
3. The Lambda expression processes **aNumber:**
    a. Use **aNumber** in some logic to modify the reducer (**sum**)
4. The reducer is "remembered" from call to call to the Lambda expression.
5. When all elements in **numbersToSum** are processed, **reduce()** returns the reducer which has the Collection object collapsed/converted to a single value.

```
numbersToSum.reduce(0, (sum, aNumber)-> {
                                    sum += aNumber;
                                    });
```

Method References

Sometimes a lambda expression does nothing but call an existing method.

In those cases, it's often clearer to refer to the existing method by name.

Method references enable you to do this; they are **compact, easy-to-read lambda expressions for methods that already have a name.**

A method reference use two colons to separate the class type or object from the method.
In this reading, we will cover three types of method references:

- Reference to an instance method of a particular object format:
    - containingObject::instanceMethodName
- Reference to an instance method of an arbitrary object of a particular type format:
    - ContainingType::methodName
- Reference to a static method format:
    - ContainingClass::staticMethodName