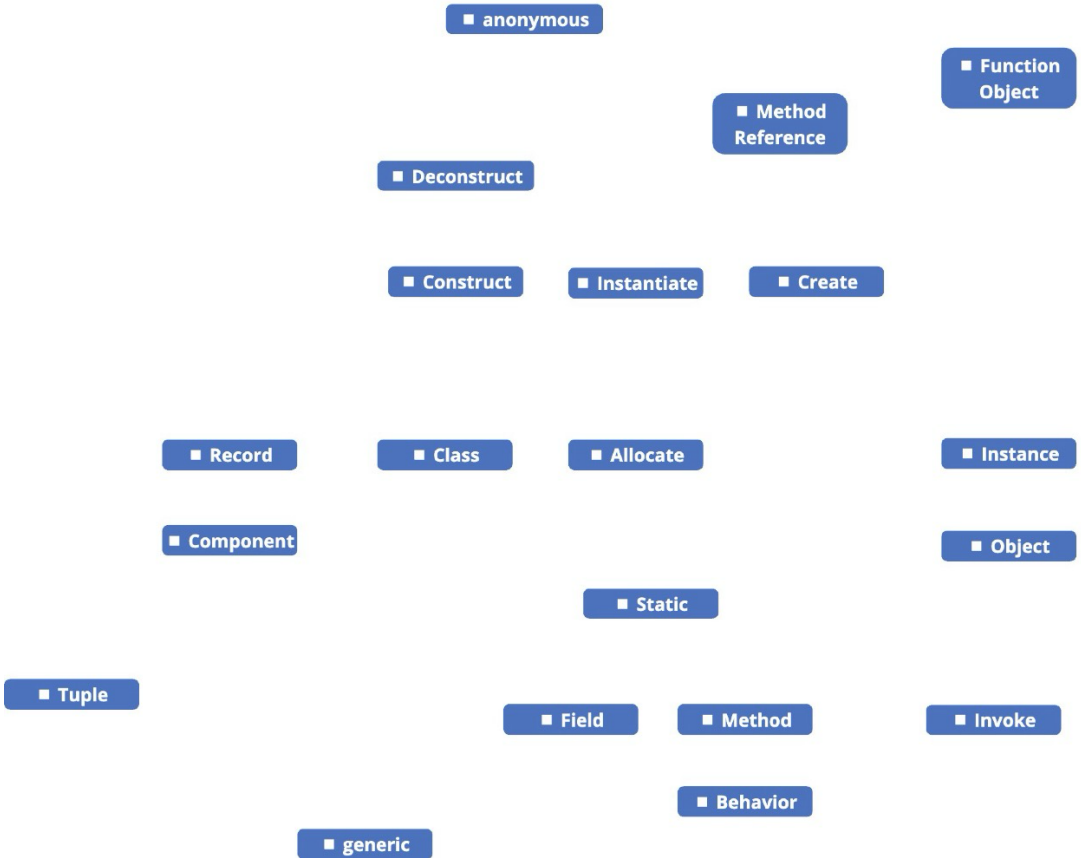
More Records, Anonymous Functions

Student name:	TA signature:

Concepts Check off understood concepts, connect related concepts, label connections

Add the following three concepts to the map: **Lambda**, **State**, and **Higher-Order Function**. Connect them and connect everything else as well.



Make sure you can **explain** each concept and each connection, you can provide **examples**, and you can **identify** them in a given piece of code.

Names Circle the methods, underline the types

There are no new names this week

Seeing Records as Table Rows

In **informatics** we deal with **information**. Often that information is stored in tables. A table has rows and columns. Here is a table with the 21 neighborhoods of Lugano (https://www.lugano.ch/la-mia-citta/identita-e-storia/quartieri.html):

Name	Population	Area [km²]	Min Elevation	Max Elevation
Barbengo	2351	2.62	370	475
Besso	5193	0.65		
Brè-Aldesago	1072	4.2		
Breganzona	5364	2.28	435	515
Cadro	2740	4.56	475	1516
Carabbia	615	1.07	518	588
Carona	949	4.69	602	911
Castagnola-Cassarate-Ruvigliana	6424	1.43		
Centro	5404	1.03		
Cureggia	178	0.66	653	1180
Davesco-Soragno	1595	2.5	421	1517
Gandria	285	3.45	292	1034
Loreto	3080	0.59		
Molino Nuovo	9662	1.23		
Pambio-Noranco	803	0.55	349	376
Pazzallo	1666	1.62	423	856
Pregassona	9400	2.26	378	1250
Sonvico	2098	11.06	603	1491
Val Colla	1340	11.06	1100	2116
Viganello	7021	1.2	283	283
Villa Luganese	557	2.21	603	1350

How could we represent this table in Java? One option is to have five "parallel" sequences, one sequence per column. **Fill in the types:**

```
public static Sequence<          > names() { return of("Barbengo", "Besso", …); }
public static Sequence<          > population() { return of(2351, 5193, …); }
public static Sequence<          > area() { return of(2.62, 0.65, …); }
public static Sequence<          > minElevation() { return of(370.0, …); }
public static Sequence<          > maxElevation() { return of(475.0, …); }
```

This organization groups the data **by column**; by **field**. All names are grouped together into one sequence; all population counts are grouped together; …

Alternatively, we could group the data **by row**; by **object**. All information of Barbengo is grouped; all information about Besso is grouped together; …

To do this kind of grouping, we can use a **record class**. Let's declare a record class to represent neighborhoods:

```
public record Neighborhood(
          name,
          population,
          area,
          minElevation,
          maxElevation
) { }
```

In the record declaration on the left and the method declarations above:

- highlight the column **names**
- fill in the **type** for each column

Do you need to use types that are **wrapper classes** in both cases?

In object-oriented programming (OOP) we like representing information as objects. In class-based languages like Java, the fields an object must contain are specified by a class. In OOP, when you see a table like the one with our neighborhoods, you immediately design a class for it in your head.

While classes are the most common way to model tables, there are other ways. We saw above that we could store the table in several sequences, one sequence per column.

Could we represent the neighborhoods as a ***sequence* of sequences**? Try!

Constructing and Deconstructing Objects

Now that we have a class to represent neighborhoods, we can create objects of that class. Here is an expression that creates the object representing Barbengo:

```
new Neighborhood("Barbengo", 2351, 2.62, 370.0, 475.0)
```

We could write a method that returns a sequence of all Lugano neighborhoods:

```
public static Sequence<Neighborhood> luganoNeighborhoods() {
  return of(
    new Neighborhood("Barbengo", 2351, 2.62, 370.0, 475.0),
    new Neighborhood("Besso", 5193, 0.65, …
  );
}
```

We use **`new` *`ClassName`*`(…)`** to create objects of a class. We can also say:

- **allocate** an object of a class
- create an **instance** of a class
- **instantiate** a class

So, we can create objects. That's great! But once we **construct** an object, what do we do with it? We can pass it to a method, and we can return it from a method:

```
public static Neighborhood id(Neighborhood hood) {
  return hood;
}
```

That's fine, but rather boring. Can we somehow **deconstruct** the object? Can we look at its **fields**? Of course! Here is how:

```
public static double elevationDifference(Neighborhood hood) {
  return hood.maxElevation() - hood.minElevation();
}
```

The `elevationDifference` method receives a neighborhood as a parameter, determines its maximum and minimum elevation, and returns their difference.

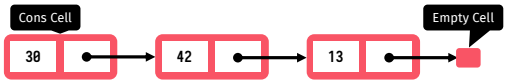
Expression Trees and Object Diagrams

We used expressions that call static methods many times before. We saw that we can call a static method **with** or **without** specifying the class name. Here are two **static method invocation** expressions, both calling the same method. **Annotate** each expression tree node with its **type**:

`luganoNeighborhoods()`	`Demo.luganoNeighborhoods()`
luganoNeighborhoods()	Demo.luganoNeighborhoods()

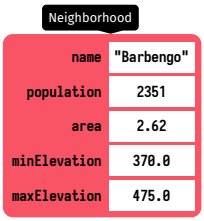
Note: The former only compiles if the expression is inside the corresponding class (in this case, inside class `Demo`), or if the method name has been imported using a static import.

We have shown object diagrams in the past, e.g., of **cons cells** and **empty cells**:



Given that we can use **record classes** to create **our own types** of objects, we also can draw those. Here is a diagram of the object of record class `Neighborhood` that represents Barbengo.

It shows a variable for each field, and next to the variable we write the field's name.



We just encountered **two important kinds of expressions**. **Class instance creation** expressions consist of the keyword **new** followed by the name of the class, and a list of constructor arguments. **Instance method invocation** expressions consist of a hole (for a subexpression producing an object), a dot, the name of the method, and a list of method arguments. **Annotate** each expression tree node with its **type**:

new Neigborhood("Barbengo", 2351, 2.62, 370.0, 475.0)	`hood.maxElevation()`
new Neighborhood(□,□,□,□,□)	□.maxElevation()

Draw the expression tree for the following expression. Annotate nodes with **types** and **values**. To show values of record classes (objects), draw the object diagram:

new Neigborhood("Carona", 949, 4.69, 602.0, 911.0).area()

Lambdas

Last week we wrote several small methods to use as mappers, predicates, or combining functions. Here are some examples:

```
public class Num {
  public static double abs(double v) {
    return v < 0 ? -v : v;
  }
  public static double inc(double v) {
    return v + 1;
  }
}
public class Compare {
  public static boolean isZero(int a) {
    return a == 0;
  }
  public static boolean gt(double a, double b) {
    return a > b;
  }
}
public class Visual {
  public static Graphic rotate180(Graphic g) {
    return rotate(180, g);
  }
  public static Graphic underlay(Graphic a, Graphic b) {
    return overlay(b, a);
  }
}
```

We then used those **functions as values** by writing a **method reference** to construct a function object:

```
Num::abs
```

This allowed us to call **higher-order functions** like `map`, `filter`, and `reduce`:

```
map(Num::inc, of(2.0, 1.0, 4.0))
```

In the following table, check each cell if the corresponding function can be used as a **mapper** in `map`, as a **predicate** in `filter`, or as a **combining function** in `reduce`:

		`map`	`filter`	`reduce`
Function	**Signature**	⟶	⟶	⟶
`abs`	Do ⟶ Do			
`inc`	Do ⟶ Do			
`isZero`	In ⟶ Bo			
`gt`	Do, Do ⟶ Bo			
`rotate180`	Gr ⟶ Gr			
`underlay`	Gr, Gr ⟶ Gr			

While it is fantastic that method references allow us to pass around functions, having to write a method just to, e.g., specify how to increment something is **painful**.

Lambdas provide a convenient alternative to method references; they are a way to create a function object right on the spot. Here is an example:

Using method reference	Using lambda

`public class Num {`
`  public static double inc(double v) {`
`  return v + 1;`
`  }`
`}`
`map(Num::inc, of(2.0, 1.0, 4.0))`
	`// no class or method needed!`
	
	`map((v) -> v + 1, of(2.0, 1.0, 4.0))`

Translate between uses of **method references** and uses of **lambdas**:

`map(`
`  Num::abs,`
`  of(2.0, -1.0)`
`)`	

`map(`
`  Compare::isZero,`
`  of(1, 0)`
`)`	

	`filter(`
	`  (v) -> v == 0,`
	`  of(0, 4, 1)`
	`)`

`map(`
`  Visual::rotate180,`
`  of(triangle(90, 90, 60, RED))`
`)`	

	`reduce(`
	`  (a, b) -> overlay(b, a),`
	`  emptyGraphic(),`
	`  of(g1, g2, g3)`
	`)`

Do this: In one of your lab repositories, create a class `Playground`, put each of the above 10 code snippets into a separate method, and test the method.

A lambda is a way to create a function right at a point where one needs one. That function has no name. It's an **anonymous function**.

Like a method, a lambda can have zero, one, or more **parameters**. And like a method, it **returns a value**. The syntax for lambdas in Java is quite flexible: you can specify the parameter types, or you can leave them out (which is quite extreme for Java!), and if you have only one parameter, you don't need to write the () around the parameter:

Method	**Lambda** (with alternatives)

`public static int five() {`
`  return 5;`
`}`
	`() -> 5`

`public static int twice(int v) {`
`  return v * 2;`
`}`
	`(int v) -> v * 2`
	`(  v) -> v * 2`
	`  v  -> v * 2`

`public static int mul(int a, int b) {`
`  return a * b;`
`}`
	`(int a, int b) -> a * b`
	`(  a,  b) -> a * b`

Another Record: ParliamentaryGroup

The Swiss parliament is split into different parliamentary groups. Each group consists of one or more political parties. Here are the parliamentary groups of the 51st legislative period from 2019 to 2023 (https://www.parlament.ch/en/organe/groups):

Name	Acronym	President	Members
Swiss People's Party	V	Thomas Aeschi	62
Social Democrats	S	Roger Nordmann	47
The Centre Group	ME	Philipp Matthias Bregy	45
FDP. The Liberal Group	RL	Damien Cottier	41
Green group	G	Aline Trede	35
Green liberal group	GL	Tiana Angelina Moser	16

Model this table as a Java **record** named `ParliamentaryGroup`:

```
public record
```

Implement the following method so it instantiates a parliamentary group of your choice (a real one from the table above, or one you make up):

```
public static ParliamentaryGroup someGroup() {
  return
}
```

Implement the following method to get the names of the presidents of all groups. Use a **lambda** in your implementation:

```
public static Sequence<String> presidentNames(
  Sequence<ParliamentaryGroup> groups)
{
  return


}
```

Implement the following method to find all groups that have fewer than the given number of members. Use a **lambda** in your implementation:

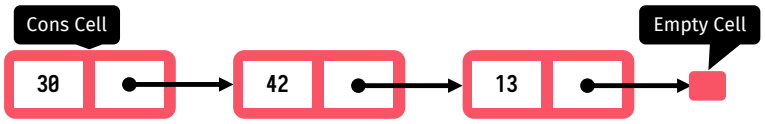
```
public static Sequence<ParliamentaryGroup> groupsSmallerThan(
  int members,
  Sequence<ParliamentaryGroup> groups)
{
  return


}
```

Pairs

You know that `Sequence<T>` is a parametric type with `T` as its type parameter. It can represent sequences of 0 or more elements of type `T`.

Here is a sequence of integers, which you could construct with `of(30, 42, 13)`:



The sequence consists of four objects: three Cons cells, and one Empty cell. Each Cons cell contains two values: the element (in this case an integer number) and the reference to the rest of the sequence (shown as an arrow).

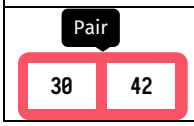
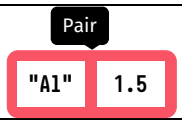
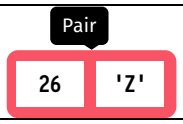
`Pair<F,S>` is another parametric type, with two type parameters: `F` and `S`. A pair can be seen as a fixed-length sequence containing exactly **two** elements: a first element of type `F`, and a second element of type `S`. You can create one like this:

```
<F,S> Pair<F,S> pair(F first, S second)
```

Here are a few examples. Complete the empty cells in the table:

`pair(30, 42)`	`pair("Hi", "ho")`		`pair(26, 'Z')`

`Pair<`
`  Integer,`
`  Integer`
`>`
	`Pair<`
	`  String,`
	`  String`
	`>`
		`Pair<`
		`  String,`
		`  Double`
		`>`	

Deconstructing Pairs

When introducing the `Sequence` class, we discussed how we can **construct** a sequence, and how we can **deconstruct** a sequence. We just saw how to **construct** a pair, using the `pair` method. Here are two methods to **deconstruct** a given pair:

```
<F,S> F firstElement(Pair<F,S> pair)

<F,S> S secondElement(Pair<F,S> pair)
```

Write a method `makePosition` that takes two parameters named `x` and `y` of type `double` and constructs and returns a position (a pair of two doubles):

```
public static
```

Write a method `getX` that takes a parameter that's a pair of two doubles and returns the x-part of the position:

```
public static
```

Zipping Sequences

We can **zip** two sequences to get a sequence of pairs. Complete the table:

`zip(of(1, 2, 3), of(4, 5, 6))`	(1 4) (2 5) (3 6)
`zip(range(1, 5), range(5, 1, -1))`	
`zip(range(3), of("A", "B", "C"))`	

A special case of zip is **zipWithIndex**:

`zipWithIndex(of(0, 0, 7))`	(0 0) (0 1) (7 2)
`zipWithIndex(of("A", "B", "C"))`	

Above we use `(… …)` to represent a pair. Java does not have literals for pairs, thus there is no such short way to represent them in text, except using an expression that creates them, e.g., `pair(…, …)`.

Which of the following statements are correct?

- ❑ You can concatenate two sequences with different element types.
- ❑ You can zip two sequences with different element types.

Composing Sequences and Pairs

Assume we want to represent the following data:

"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"
1	2	3	4	5	6	7	8

We can represent this in two ways:

Sequence of pairs `Sequence<Pair<String,Integer>>`:

"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"
1	2	3	4	5	6	7	8

Write an expression using `pair` and `of` to compose the above sequence of pairs:

Pair of sequences `Pair<Sequence<String>,Sequence<Integer>>`:

"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"
1	2	3	4	5	6	7	8

Write an expression using `pair` and `of` to compose the above pair of sequences:

Note: We can use `zip` to convert a **pair of sequences** into a **sequence of pairs**.

The Two Fundamental Aspects of an Object

Objects are instances of classes. Objects combine two fundamental aspects.

Complete this table:

Instance field	**Instance method**
Field inside an object	Method working on an
Object's **state**	Object's
	`expr.method()`
	

Instance Fields and Instance Methods of Record Classes

Records are instances of record classes.

Any record has an **instance field** and **instance method** for each **component**. These methods return the value of the corresponding instance field. The method has the same name as the field, has no parameters, and has a return type that is the same as the type of the field.

Complete this list of instance fields and instance methods:

Record (Class)	**Instance fields**	**Instance methods**

`Neighborhood`	`name`
	`population`
	`area`
	`minElevation`
	`maxElevation`	

`ParliamentaryGroup`		

Sorting

Can you recognize the following sorting algorithm?

```
public static <T> Sequence<T> sort(
  Function2<T,T,Boolean> lessEqual,
  Sequence<T> sequence
) {
  return isEmpty(sequence)
    ? empty()
    : concat(
        sort(
          lessEqual,
          filter(x -> lessEqual.apply(x, first(sequence)), rest(sequence))
        ),
        cons(
          first(sequence),
          sort(
            lessEqual,
            filter(x -> !lessEqual.apply(x, first(sequence)), rest(sequence))
          )
        )
    );
}
```

Explain how this code corresponds to the known sorting algorithm:

The `sort` method is generic. We can sort a sequence of any type of element. We just need to provide a comparison function for the parameter `lessEqual`, which the algorithm uses to tell whether an element is less or equal to another element.

Let's sort a sequence of **integers** and a sequence of **doubles**:

```
sort((a, b) -> a <= b, range(100, 0, -1))
sort((a, b) -> a <= b, of(1.5, 2.0, 0.1))
```

Complete this expression to sort a sequence of **booleans** (such that `false` is considered less than `true`):

```
sort(
  (a, b) ->
  of(true, false, false, true, false)
)
```

To sort a sequence of **strings**, we need a way to compare two strings. The `String` class has an **instance method** `compareTo` which does this: `"A".compareTo("B")` evaluates to -1, `"A".compareTo("A")` to 0, and `"B".compareTo("A")` to 1.

Complete this expression to sort a sequence of strings in alphabetical order:

```
sort(
  (a, b) ->
  of("Hello", "Ciao", "Hallo", "Salut")
)
```

Let's sort some more interesting data. Also **check your code** in an IDE!

Complete this expression to sort a sequence of neighborhoods by population:

```
sort(
  (a, b) ->
  luganoNeighborhoods()
)
```

Complete this expression to sort a sequence of neighborhoods by name, in reverse alphabetical order:

```
sort(
  (a, b) ->
  luganoNeighborhoods()
)
```

Assume a method `parliamentaryGroups()` that returns a sequence with the Swiss parliamentary groups (as specified on a prior page). **Complete this** expression so it sorts that sequence by acronym:

```
sort(
  (a, b) ->
  parliamentaryGroups()
)
```

Complete this expression to sort a sequence of pairs by their first element:

```
sort(
  (a, b) ->
  of(
    pair(1, "X"),
    pair(9, "A"),
    pair(3, "Z"),
  )
)
```

Write an expression to get the sequence of names of the top-10 Lugano neighborhoods by area. Use `luganoNeighborhoods`, `sort`, `take`, and `map`:

Write an expression to get the sequence of names of Lugano neighborhoods ordered by their elevation difference. Filter out neighborhoods for which the elevation is not known (which you can detect if `Double.isNaN` returns `true` for `minElevation` or `maxElevation`). Use `luganoNeighborhoods`, `filter`, `sort`, and `map`:

Pair is a Record Class

Pairs are objects of type `Pair`. We **constructed** pairs by calling `Pairs.pair`:

```
pair("Al", 1.5)          -or-          Pairs.pair("Al", 1.5)
```

What does this method do to create a pair object? Here is the implementation:

```
public static <F,S> Pair<F,S> pair(F a, S b) {
  return new Pair<F,S>(a, b);
}
```

It creates an instance of class `Pair` the same way we created an instance of class `Neighborhood` or `ParliamentaryGroup`. The only difference is the type parameters. Pairs are generic (they have type parameters). A pair has two fields, but the type is not baked into the `Pair` class; instead it can be chosen whenever a pair is created.

The following two expressions produce the same result:

`Pairs.pair(12, "Hi")`	`new Pair<Integer,String>(12, "Hi")`

When we **deconstructed** pairs, we used `firstElement` and `secondElement`:

```
Pairs.firstElement(pair(12, "Hi"))
```

What do those methods do? Here is the implementation of `firstElement`:

```
public static <F,S> F firstElement(Pair<F,S> pair) {
  return pair.first();
}
```

Given this information, **write down** the source code of the `Pair` record class:

```
public record
```

Note: Because `Pair` is a **generic class**, in the above definition you have to introduce the type parameters. This is similar to introducing type parameters for a **generic method** (we have to write something like `<A,B>`). Guess how to do that. Then **do this**: check that you are correct by creating your own `Pair` class in an IDE.

Now that you had a look behind the scenes, in theory you don't need class `Pairs` and its static methods (`pair`, `firstElement`, and `secondElement`) anymore.

Reimplement the following expressions without using class `Pairs`:

```
firstElement(pair(1, 2))
```

```
new Pair<Integer,Integer>(1, 2).first()
```

```
pair(firstElement(pair(1, 'a')), secondElement(pair("A", 1.4)))
```

Record Types for Other Kinds of Tuples

A pair is a **tuple** with **two** fields. **Declare** record classes for triples and quadruples:

`public record Triple<A,B,C>(`	`public record`

Going to the Limit

Let's declare a record class with a single field. It doesn't really do anything. It just wraps an object around the given value.

```
public record Monuple<T>(T t) {
}
```

This is similarly “useless” to creating a function that just returns whatever it receives as a parameter. That function is called "identify function":

```
public static <T> T id(T t) {
  return t;
}
```

While a `Monuple` record can "store" a value, the `id` function can "process" a value.

Let's go even crazier. Can you **declare** a record `EmptyTuple` without any field at all?

Do this: In an IDE, declare record classes `EmptyTuple`, `Monuple`, `Pair`, `Triple`, and `Quadruple`, and a class `Fun` containing method `id`, and make sure they work as expected.