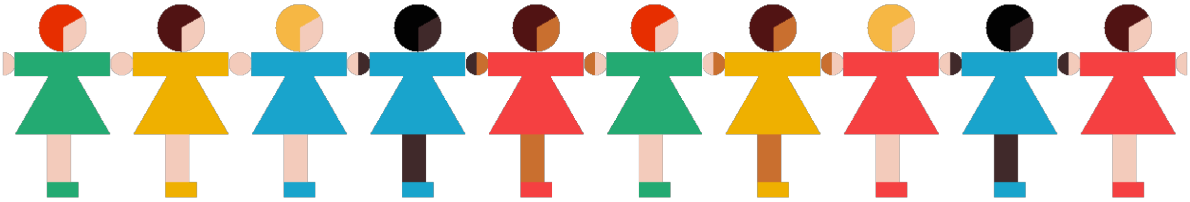
Repetitive Computation

Student name:

TA signature:



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

Add the following three concepts to the map: **Recursion**, **Method**, and **Expression Tree**. Connect them and connect everything else as well.

- ☐ Class
- ☐ Stack Overflow
- ☐ Stack Frame
- ☐ Library
- ☐ Call Stack
- ☐ Call Site
- ☐ Value
- ☐ JTamaro Library
- ☐ Java Library
- ☐ Termination Condition
- ☐ Type Signature
- ☐ Call
- ☐ Return
- ☐ Overloading
- ☐ Infinite Recursion
- ☐ Base Case
- ☐ Recursive Case
- ☐ Operator
- ☐ Expression
- ☐ Operand
- ☐ Arity
- ☐ Dynamic Call Tree
- ☐ Map
- ☐ Filter
- ☐ Reduce
- ☐ Predicate
- ☐ Neutral Element
- ☐ Depth-First Traversal

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

`Math` • `min` • `max` • `pow` • `sqrt` • `Integer` • `parseInt` • `Double` • `parseDouble` • `Boolean` • `parseBoolean` • `hsv` • `hsl` • `intersperse` • `concat`

Calls and Returns Language

The place in a code where we call something is a call site. When you call a method, its body gets executed, and then it returns to the call site. In the following code, each call site is highlighted in yellow:

```
public static Graphic eye(double diameter) {
    return overlay(
        Toolbelt.circle(diameter * 0.5, BLACK),
        Toolbelt.circle(diameter, WHITE)
    );
}

public static Graphic eyes(double diameter) {
    return beside(
        eye(diameter),
        eye(diameter)
    );
}
```

Which claims about the execution of the above code are correct, assuming the execution starts with the call `eyes(100)`?

True	False	Claim
☐	☐	Method `beside` gets called **before** method `eye` gets called
☐	☐	Method `beside` returns **before** any of the calls to method `eye` return
☐	☐	Method `beside` returns **after** all of the calls to method `eye` return
☐	☐	Method `beside` gets called **after** all of the calls to method `eye` return
☐	☐	Method `overlay` gets called **after** method `beside` gets called
☐	☐	Method `Toolbelt.circle` gets called **after** method `eye` gets called
☐	☐	Method `Toolbelt.circle` gets called twice
☐	☐	Method `eye` gets called twice
☐	☐	Method `overlay` gets called twice
☐	☐	Including the call to `eyes`, a total of 10 method calls happen

Each call is paired with a return. And calls and returns are nested.

Dynamic Call Trees Language

Because call-return pairs are properly nested, we can represent the sequence of calls and returns being executed as a tree. That is the dynamic call tree.

In a dynamic call tree, each call-return pair (we often just say "each call") is represented by a **node**. The nodes are labeled with the **name of the method** and often also with the **values of the arguments** for that call. The root of the call tree represents the method we initially call.

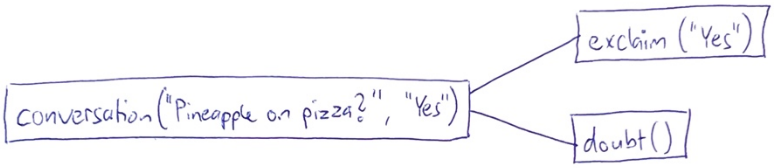
The dynamic call tree provides a global picture of what **happens during execution**. It focuses on method **calls** and **returns**. It does not show what happens inside each method, except that it calls some methods. It does not show operators being evaluated. It is a tree, but it is **very different** from an expression tree.

In this course we draw dynamic call trees from left to right (root on the left).

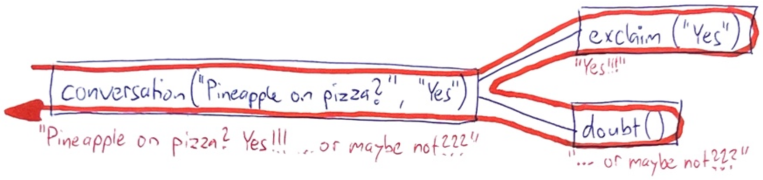
Here are three example methods:

```
public static String doubt() {
    return " ... or maybe not???";
}

public static String exclaim(String statement) {
    return statement + "!!!";
}

public static String conversation(String question, String answer) {
    return question + " " + exclaim(answer) + doubt();
}
```

Below is the dynamic call tree for the call:
`conversation("Pineapple on pizza?", "Yes")`



The program execution corresponds to the depth-first traversal of the **call tree**. We can visualize that traversal with a line. We also can show the return values flowing back (i.e., towards the left) when returning from each call.



There is **a return paired with every call** (unless the program "crashes" before the call returns).
Given the `add` and `sum` methods below, draw the dynamic call tree of the call `sum(1, 2, 3, 4)`, show the traversal with a line, and show the return value of each call.

```
public static int add(int a, int b) {
    return a + b;
}

public static int sum(int a, int b, int c, int d) {
    return add(add(a, b), add(c, d));
}
```

Given the `eye` and `eyes` methods from before, draw the dynamic call tree of the call `eyes(100)`, show the traversal with a line, and show the return value of each call.

Dynamic Call Tree vs. Expression Tree Language

How does a dynamic call tree differ from an expression tree?

Call Tree	**Expr. Tree**	**What?**
		Shows a single expression
		Shows **static** information (is not about one specific execution)
		Shows **dynamic** information (is about one specific execution)
		Shows all operations
		Shows only calls/returns
		Can show information across multiple methods
		Shows all the details corresponding to one expression

An expression tree is a tree representing **one** expression as written in source code. A dynamic call tree is a tree representing **all** calls in an execution.

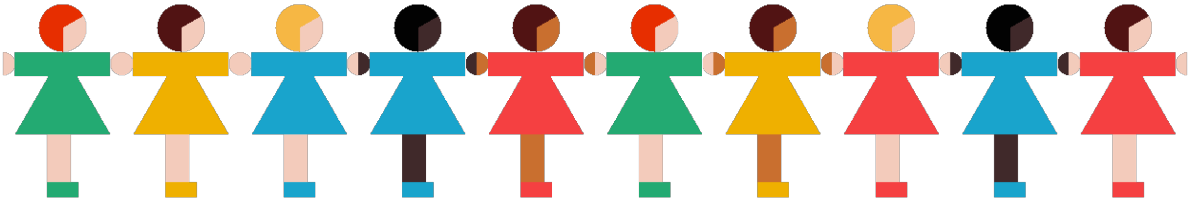
Watch out! Do not confuse them!
They are both trees, and they both include method calls, but they are fundamentally different. To help you, we draw:

- *Dynamic call trees* horizontally, from the left to the right
- *Expression trees* vertically, from the top to the bottom

Amusement Park Line

Language

You get to your favorite ride in an amusement park, but there is a very long line. You'd like to know how many people are in front of you in the line, so that you can estimate how long you will have to wait.



You could walk along the entire line, to count each person. Or you could be lazy. How can you find out how long the line is with minimal effort?

Sure, you could send a friend to walk along the whole line and to count everyone. But all your friends want to be lazy, too. You come up with a clever plan: you ask the person on your right (in the above picture) how many people are in the line that starts with them and goes all the way to the right. Assume that person is lazy as well. How will **they** figure it out? They will ask the same question to the person on their right. And so on and so on. Until the question reaches the person at the right edge. There is nobody in front of that person. So that person answers 1. The person on their left computes 1 (themselves) + 1 (the answer obtained from the person on their right), and answers 2. And so on.

If we translate this to code, and we represent a line of persons as a `Sequence<Person>` (assuming there is a class `Person` somewhere):

```
public static int count(Sequence<Person> persons) {
  return 1 + (                      // me +
    isEmpty(rest(persons))          // nobody on my right?
      ? 0                             // nobody (on my right)
      : count(rest(persons))          // rest of persons (on my right)
  );
}
```

What happens if we call `count(empty())`?

Improve the `count` method so that it works for empty sequences as well:

```
public static int count(Sequence<Person> persons) {





}
```

Recursive Methods Language

The methods on the previous page are recursive; they call **themselves**. E.g., method `count` calls... method `count`. You see that call in the body of the method:

```
public static int count(Sequence<Person> persons) {
  return isEmpty(persons)         // termination condition
    ? 0                           // base case
    : 1 + count(rest(persons));   // recursive case
}
```

The recursive **call site** is in the recursive case of the conditional. The **condition** in the conditional expression serves as a termination condition: it determines whether you reached the base case of the recursion.

Assume that `al` and `ed` are names of values of type `Person`. Draw the dynamic call tree of `demo(al, ed)`, including the traversal line and return values:

```
public static int demo(Person a, Person b) {
  return count(cons(a, cons(b, empty())));
}
```

Here is another recursive method:

```
public static int eternity() {
  return eternity();              // recursive case
}
```

What happens when you run this?

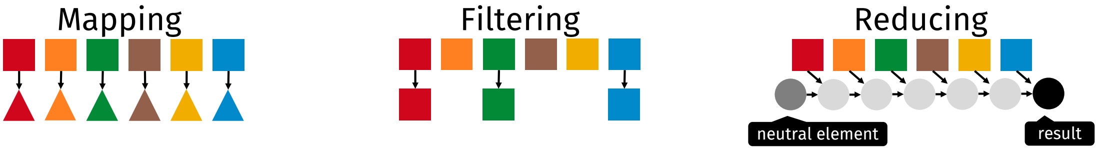
In theory, calling this method would lead to an infinite recursion. Every call of the method would lead to another call of the method. None of the calls would ever return.

In practice, in most programming languages, calling this method will eventually "crash" the program. Why? Because every call will allocate a new stack frame (a piece of memory necessary for the method to execute) on the call stack and given that we just keep calling and never get to return, eventually we will run out of memory for our call stack. In Java, this leads to a stack overflow.

The method does **not** have a termination condition. It only has a recursive case—there's really no "case"; there's no decision with multiple possibilities; it's always just going to call itself.

Processing Sequences Language Library

The probably most common tasks with a sequence are:



All tasks start with a given sequence of some type (■) of elements.

Mapping Sequences

When we map, we want to transform (i.e., *to map*) each element of the given sequence into some other element. The result is a sequence with a potentially different type (▲) of elements. Here is an example to map from colors to colored dots:

```
public static Sequence<Graphic> colorsToDots(Sequence<Color> colors) {
  return isEmpty(colors)                       // termination condition
    ? empty()                                  // base case
    : cons(                                    // recursive case
        Toolbelt.circle(100, first(colors)),       // map 1 element
        colorsToDots(rest(colors))                 // map rest
      );
}
```

When mapping from one sequence to another sequence, we essentially map each element into something else. Thus, we can implement the **mapping of an individual element** as a separate method:

```
public static Sequence<Graphic> colorsToDots(Sequence<Color> colors) {
  return isEmpty(colors)                       // termination condition
    ? empty()                                  // base case
    : cons(                                    // recursive case
        colorToDot(first(colors)),                 // map 1 element
        colorsToDots(rest(colors))                 // map rest
      );
}
public static Graphic colorToDot(Color color) {
  return Toolbelt.circle(100, color);              // map 1 element
}
```

Write a method that maps an angle into a black size-20 square rotated by angle:

```
public static Graphic angleToR(int angle) {
  return
}
```

Write a method that maps a sequence of angles into a sequence of squares rotated by those angles:

```
public static Sequence<Graphic> anglesToRs(Sequence<Integer> angles) {
  return



}
```

Write a method that maps a sequence of numbers into a sequence of negated numbers (multiply each number by -1). Note that here the element type **is the same** for the sequence passed as an argument and the returned sequence:

```
public static Sequence<Double> negate(Sequence<Double> numbers) {
  return






}
```

Write a method that maps a sequence of strings into a sequence of integers, using the method `Integer.parseInt` to map an individual string to an integer:

```
public static Sequence<Integer> parseInts(Sequence<String> strings) {
  return






}
```

Filtering Sequences

When we filter, we want to keep only elements that fulfill a certain condition (the predicate). The result is a sequence of the same type (■) of elements. Here is an example to get all the positive numbers of the given sequence:

```
public static Sequence<Integer> positives(Sequence<Integer> numbers) {
  return isEmpty(numbers)                     // termination condition
    ? empty()                                 // base case
    : (                                       // recursive case
      first(numbers) >= 0                                // predicate
        ? cons(first(numbers), positives(rest(numbers))) // keep
        : positives(rest(numbers))                       // drop
      );
}
```

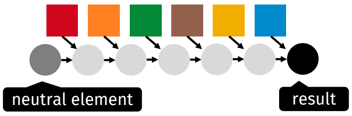
This method **deconstructs** the given sequence and **constructs** the resulting sequence. The recursive case handles a non-empty sequence. It looks at the filter condition to determine whether to produce a sequence with or without the current element.

Write a method that gets all the non-empty (non-zero-length) strings from the given sequence, assume there is a method `len(String)` you can use as predicate:

```
public static Sequence<String> nonEmpty(Sequence<String> strings) {






}
```

Reducing Sequences

When we reduce a sequence, we combine all the elements of the given sequence into **one thing** of a possibly different type (●).



To reduce, we start with an initial value, the so-called neutral element. Then we combine that value with the first element of the sequence. Then we combine that intermediate result of that with the second element of the sequence, and so on, until we combine the intermediate result with the last element of the sequence and end up with the final **result**.

Here is an example reduction, to compute the product of a sequence of numbers:

```
public static Integer product(Sequence<Integer> numbers) {
  return isEmpty(numbers)
    ? 1
    : first(numbers) * product(rest(numbers));
}
```

Here is another example reduction, to join a sequence of strings:

```
public static String join(Sequence<String> strings) {
  return isEmpty(strings)
    ? ""
    : first(strings) + join(rest(strings));
}
```

Complete this third example reduction, to put above a sequence of graphics:

```
public static Graphic aboves(Sequence<Graphic> graphics) {
  return isEmpty(graphics)
    ?
    :
}
```

Complete the following table to summarize the three reductions seen so far:

Reduction	**Type of element (■):**	**Neutral element:**	**Combining operation:**	**Type of result (●):**
`product`				
`join`				
`aboves`				

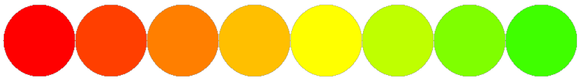
A value `I` is a **neutral element** for a binary operation ⊕, if `a ⊕ I = a = I ⊕ a`. The neutral element of the `*` is 1, because multiplying something by 1 doesn't change it. The same idea applies to `+` for strings and `above` for graphics.

In reductions, the neutral element, all the intermediate results, and the final result have the same type (●). However, the type of the elements of the sequence (■) does not need to be the same.

A whole pipeline using filter, map, and reduce

Let's build a **pipeline** to turn a sequence of angles into a graphic, by filtering, mapping, and reducing sequences. First, create a sequence of angles. Then filter that sequence to eliminate illegal angles (outside the interval [0, 360[). Then turn that sequence of legal angles into a sequence of colors (where the color's hue corresponds to the angle). Then turn that sequence of colors into a sequence of graphics (colored dots), and then reduce that sequence of graphics into a single graphic (place them beside each other).

```
besides(              // reduce
  colorsToDots(       // map
    anglesToColors(   // map
      legalAngles(    // filter
        angles()      // generate
      )
    )
  )
)
```



Expression Tree

Draw the expression tree of the above expression.

Produce a list of angles

```
public static Sequence<Integer> angles() {
  return range(-120, 120, 15);
}
```

Filter the list of angles, keeping only the legal ones

```
public static Sequence<Integer> legalAngles(Sequence<Integer> numbers) {
  return isEmpty(numbers)
    ? empty()
    : (
        first(numbers) >= 0 & first(numbers) < 360
        ? cons(first(numbers), legalAngles(rest(numbers)))
        : legalAngles(rest(numbers))
      );
}
```

Map one angle into one color (with angle as hue)

```
public static Color angleToColor(int angle) {
  return hsv(angle, 1, 1);
}
```

Map a sequence of angles into a sequence of colors (with angles as hues)

```
public static Sequence<Color> anglesToColors(Sequence<Integer> angles) {
  return isEmpty(angles)
    ? empty()
    : cons(
        angleToColor(first(angles)),
        anglesToColors(rest(angles))
      );
}
```

Map one color into one dot

```
public static Graphic colorToDot(Color color) {
  return Toolbelt.circle(100, color);
}
```

Map a sequence of colors into a sequence of graphics (colored dots)

```
public static Sequence<Graphic> colorsToDots(Sequence<Color> colors) {
  return isEmpty(colors)
    ? empty()
    : cons(
        colorToDot(first(colors)),
        colorsToDots(rest(colors))
    );
}
```

Reduce a sequence of graphics into a single graphic (with beside)

```
public static Graphic besides(Sequence<Graphic> graphics) {
  return isEmpty(graphics)
    ? emptyGraphic()
    : beside(
        first(graphics),
        besides(rest(graphics))
      );
}
```

Map, filter and reduce are common patterns of computation. In a future workbook, we will **abstract** over these patterns, so that they can be implemented once for all and then conveniently used.

Operators Language

We encountered quite a few operators. Here is a summary, and a few important new ones, grouped by their arity (number of operands):

Unary Operators (One Operand)

Arithmetic

Operator	Type Signature	Description
+	int ⟶ int	
	double ⟶ double	
-	int ⟶ int	Negation
	double ⟶ double	Negation

Logical

Operator	Type Signature	Description
!	boolean ⟶ boolean	Negation

Binary Operators (Two Operands)

Arithmetic

Operator	Type Signature	Description
+	int, int ⟶ int	Addition
	double, double ⟶ double	Addition
-	int, int ⟶ int	Subtraction
	double, double ⟶ double	Subtraction
*	int, int ⟶ int	Multiplication
	double, double ⟶ double	Multiplication
/	int, int ⟶ int	Division
	double, double ⟶ double	Division
%	int, int ⟶ int	Remainder

Logic

Operator	Type Signature	Description
&	boolean, boolean ⟶ boolean	And
&&	boolean, boolean ⟶ boolean	And (short-circuit)
|	boolean, boolean ⟶ boolean	Or
||	boolean, boolean ⟶ boolean	Or (short-circuit)

String

Operator	Type Signature	Description
+	String, String ⟶ String	Concatenation

Comparison

Operator	Type Signature	Description
<	int, int ⟶ boolean	Less than
	double, double ⟶ boolean	Less than
<=	int, int ⟶ boolean	Less than or equal
	double, double ⟶ boolean	Less than or equal
==	int, int ⟶ boolean	Equal
>=	int, int ⟶ boolean	Greater than or equal
	double, double ⟶ boolean	Greater than or equal
>	int, int ⟶ boolean	Greater than
	double, double ⟶ boolean	Greater than

Ternary Operators (Three Operands)

Conditional

This operator exists for any type `T`.

Operator	Type Signature	Description
`? :`	`boolean, T, T` ⟶ `T`	If...Then...Else

For example, when the *then* and *else* branches are `Graphic`-producing expressions, the type signature of the conditional operator is

`boolean, Graphic, Graphic` ⟶ `Graphic`

Does it make sense to use `boolean` for `T`, like in the following expression?

```
condition ? true : false
```

We can see that some operators are overloaded. For example, there is a version of < for ints and a version for doubles.

Overloading means you have one name (or symbol, or operator, ...) that can **mean multiple different things**.

Methods Library

Besides many operators, we also encountered quite a few methods. Methods are bundled into classes, and classes are packaged into libraries.

Methods from the Java Library

The following are some methods provided as part of Java; they are part of the Java library, which you get when you install Java:

Method	Type Signature	Description
`Math.min`	`int, int` ⟶ `int`	Minimum
	`double, double` ⟶ `double`	Minimum
`Math.max`	`int, int` ⟶ `int`	Maximum
	`double, double` ⟶ `double`	Maximum
`Math.pow`	`double, double` ⟶ `double`	Power x^y
`Math.sqrt`	`double` ⟶ `double`	Square root √
`Integer.parseInt`	`String` ⟶ `int`	Convert String to int
`Double.parseDouble`	`String` ⟶ `double`	...to double
`Boolean.parseBoolean`	`String` ⟶ `boolean`	...to boolean

Method `Math.min` is overloaded: there is a version of the method for ints, and another version of the method for doubles. `Math.max` is overloaded as well.

Note that in Java, all methods are defined in some class. When we write `Math.min`, we mean the method named `min` in the class named `Math`.

To call a method, we either write `methodName()` or `ClassName.methodName()`. The former ***only*** works when the call site is inside the **same** class as the method we call, or when we use a static import (like we do in labs; we will explain that later).

Methods from the JTamaro Library

Many of the methods we used are not part of the **Java library**, but they were written by us and packaged in the JTamaro library. Complete the following table:

Method	**Type Signature**
`Graphics.rectangle`	⟶
`Graphics.triangle`	⟶
`Graphics.ellipse`	⟶
`Graphics.circularSector`	⟶
`Graphics.text`	⟶
`Graphics.emptyGraphic`	⟶
`Graphics.rotate`	⟶
`Graphics.overlay`	⟶
`Graphics.above`	⟶
`Graphics.beside`	⟶
`Graphics.compose`	⟶
`Graphics.pin`	⟶
`Colors.rgb`	⟶
`Colors.hsv`	⟶
`Colors.hsl`	⟶
`Sequences.empty`	⟶
`Sequences.cons`	⟶
`Sequences.first`	⟶
`Sequences.rest`	⟶
`Sequences.isEmpty`	⟶
`Sequences.of`	⟶
`Sequences.replicate`	⟶
`Sequences.range`	⟶
`Sequences.intersperse`	⟶
`Sequences.concat`	⟶

Note: `Color.hsv` and `hsl` were introduced in slides in Week 4, Lesson 1. `Sequences.intersperse` and `concat` were introduced in Lab 3.

Describe the pattern you see in terms of the **classes** the methods are in, and the **type signatures** of the methods (there are a couple of exceptions to the pattern):

The type signatures of **operators** and **methods** are **enormously helpful**! They guide you in plugging together expressions. Always look at the types to see what can be composed!