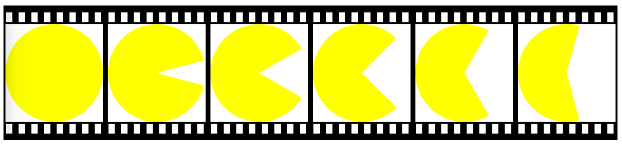
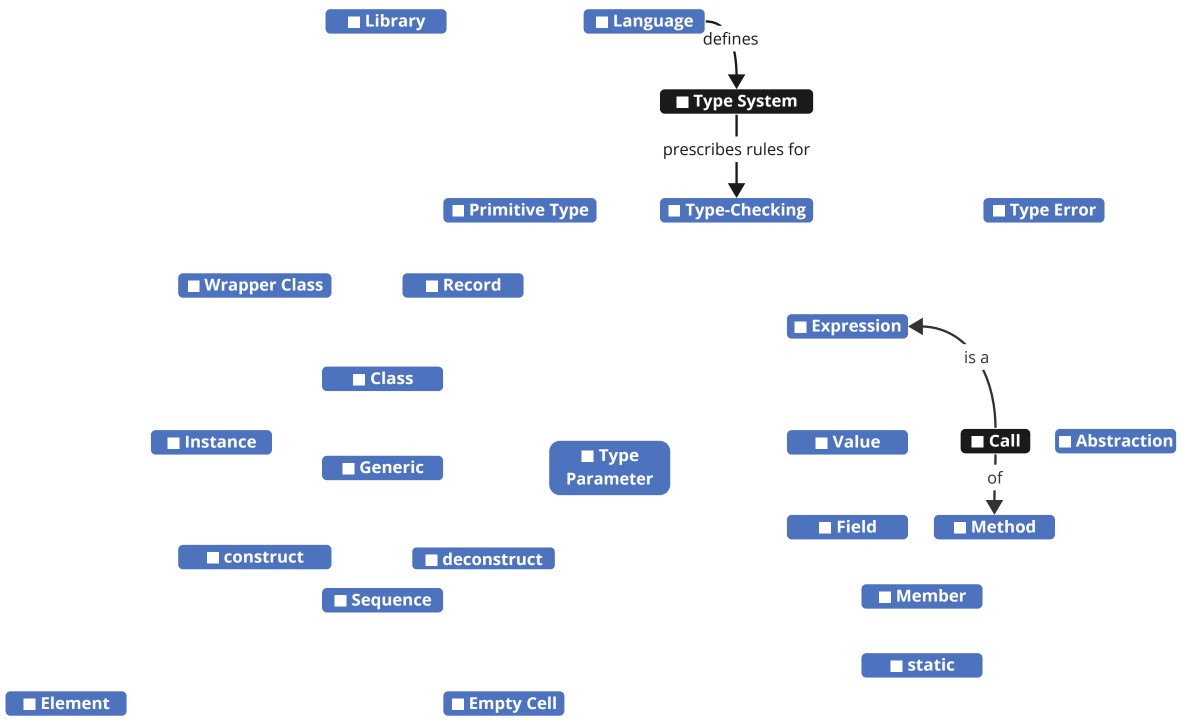
Repetitive Data

Student name:	TA signature:



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

Note: Add the following three concepts to the map: Type, Object, and Cons Cell. 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

`String` • `Character` • `Integer` • `Double` • `Boolean` • `Sequence` • `empty` • `cons` • `first` • `rest` • `isEmpty` • `of` • `replicate` • `range`

Abstraction – Turn Differences into Parameters Language

Code duplication is bad. If we see two pieces of code that are the same, we should eliminate one. But what if the two pieces have a lot of **similarities**, but do have a few **differences**?

Throughout this course we will learn to abstract over three different kinds of things: over **expressions**, over **types**, and over **behaviors**. For now, we focus on abstraction over expressions.

Abstraction over Expressions

Here is some code that builds two houses side-by-side:

```
beside(redHouse(), blackHouse())
```

And here are two methods that construct a house:

```
public static Graphic redHouse() {
  return above(
    triangle(200, 200, 60, RED),
    rectangle(200, 200, RED)
  );
}
```

```
public static Graphic blackHouse() {
  return above(
    triangle(200, 200, 60, rgb(0, 0, 0)),
    rectangle(200, 200, rgb(0, 0, 0))
  );
}
```

Do you feel the pain? You should! So many similarities, and so few differences! Highlight the differences in the above code.

Let's eliminate the code duplication. Replace the two methods above with a single method that can deal with both requests. To do this, simply turn the differences (between the two methods) into parameters:

```
public static Graphic
```

Now use your new method to build the two side-by-side houses:

Do you feel the joy? That's the joy of abstraction!

In this specific example, you abstracted over different expressions. You wrote a method that can construct a house of ***any*** color. The specific color can be provided to the method via its parameter. This is much more generally usable. You (or others) can now use your method to easily build all kinds of colorful houses!

Primitive Types vs. Classes Language

Some types are primitive types. They are baked into the Java language. In Java, you cannot create your own primitive types. In Java, the names of primitive types all start with a lowercase letter.

Other types are classes. They are part of some library. You can create your own types, because you can create your own classes. For example, the type `String` is part of the Java standard library (the Java API). The types `Color`, `Graphic`, and `Sequence` are not part of the Java standard library; they are part of the JTamaro library we created for this course. By convention, Java developers use uppercase letters as the first character of class names.

Complete the "primitive" column in the table in Workbook 2.

Wrapper Classes

In Java some types are primitive types, other types are classes. For reasons we will later see, for each primitive type Java provides a corresponding class (known as a wrapper class):

Wrapper Class	Primitive Type
`Character`	`char`
`Integer`	`int`
`Double`	`double`
`Boolean`	`boolean`

Java automatically converts between wrapper classes and the corresponding primitive types. For example, if a parameter has type `Integer`, you can pass an argument of type `int`; or if a parameter has type `char`, you can pass an argument of type `Character`.

Assume the following methods:

```
public static int one() { … }
public static Integer two() { … }
public static Integer add(Integer a, Integer b) { … }
public static int subtract(int a, int b) { … }
```

Draw an expression tree of the following expression. Annotate nodes with types:

```
subtract( add(one(), two()), subtract(two(), one()) )
```

Creating Our Own Classes Language

Besides classes like `Character`, `Integer`, `Double`, `Boolean`, and `String`, that are part of the Java library, developers can create their own classes. Let's assume we want to model vectors in two-dimensional space. We could define a class for that. Let's name it `Vector`:

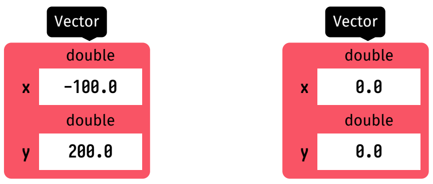
```
record Vector(double x, double y) {
}
```

The above `Vector` is a special kind of class: a record. A record is a convenient way to define a class, without having to write much code. The above definition states that `Vector`s are objects containing two fields: two `double` values named `x` and `y`.

Given the above definition, we can now construct values of that class:

```
new Vector(-100.0, 200.0)
new Vector(0.0, 0.0)
```

Those values are objects. We created two objects of type `Vector`. We can draw the two objects as follows:



Given an object, we can deconstruct it (access its fields). Records automatically define methods to get the values of their fields (these methods are known as getters):

```
new Vector(100.0, 200.0).x()
```

We say that "objects are instances of classes". For example, the object created by the expression `new Vector(1.0, 2.0)` is an instance of class `Vector`.

Because classes are types, we can write methods that take `Vector`s as parameters and return a `Vector` as a return value. **Complete** the given methods:

```
public class Vectors {

  public static Vector add(Vector a, Vector b) {
    return new Vector(a.x() + b.x(), a.y() + b.y());
  }

  public static Vector scalarMultiply(double s, Vector a) {
    return new Vector(a.x() * s, a.y() * s);
  }

  public static double dotProduct(Vector a, Vector b) {
    return
  }

}
```

Static Method Declarations

Language

In Java, methods have to be declared inside classes. We say a method is a member of a class. Here are a few classes with some static methods you may have seen:

```
public class Toolbelt {
  public static String firstName() {…}
  public static Graphic beside3(Graphic g1, Graphic g2, Graphic g3) {…}
}
```

```
public class Demo {
  public static Color purpleColor() {…}
}
```

```
public class Heart {
  public static Color purpleColor() {…}
}
```

```
public class DoubleBlackDiamond {
  public static Graphic blackDiamond(double side) {…}
  public static Graphic doubleBlackDiamond(double side) {…}
}
```

```
public class Graphics {
  public static Graphic rectangle(double w, double h, Color c) {…}
  public static Graphic beside(Graphic left, Graphic right) {…}
}
```

```
public class Colors {
  public static Color rgb(int red, int green, int blue) {…}
}
```

In the above code we highlight the `ClassName` in each class declaration, and the `methodName` in each method declaration.

If we write `purpleColor()`, which method would be called?

Static Method Invocations

Language

In the past 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 result **type**:

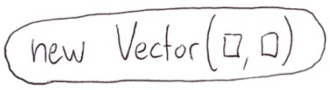
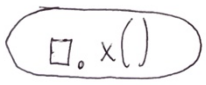
`rgb(…, …, …)`	`Colors.rgb(…, …, …)`
rgb(□, □, □)	Colors.rgb(□, □, □)

Note: The former only compiles if the expression is inside the corresponding class (in this case, inside class `Colors`), or if the method name has been imported using a static import. We will discuss the details of imports in the future.

Class Instance Creations, Instance Method Invocations

Language

When introducing record classes above, we encountered **two new 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 result **type**:

new Vector(…, …)	….x()
	

Notice how **instance** method invocations differ from **static** method invocations?

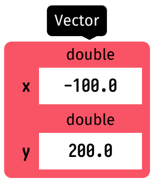
Draw the expression tree for the following expression. Annotate nodes with **types** and **values**. To show values of record classes (objects), draw the object diagram:

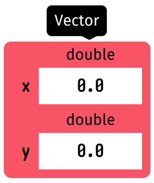
Vectors.scalarMultiply(**new** Vector(1.0, 2.0).x(), **new** Vector(3.0, 4.0))

Looking Inside Objects with Object Diagrams

Language

Let's look at how we visualize objects…





The red rounded rectangle shows the object. The black label at the top shows the type of the object (`Vector`). An object may contain fields, which are shown as white rectangles inside the object. Each field has a **name** (on the left, e.g., `x`), a type (above, e.g., `double`), and a value (inside the white rectangle, e.g., `200.0`).

```
public record Interval(int start, int end) {
}
public static Interval move(
  Interval i, int shift) {
  return new Interval(
    i.start() + shift,
    i.end() + shift
  );
}
```

Draw the object diagram of the result of:
`move(new Interval(1,2), 3)`

Sequence – A Recursive Data Structure Library Language

A **sequence** is similar to a list. We can have a sequence of integers, or a sequence of strings, or a sequence of some other type of element. A sequence can be empty (containing zero elements), it can contain one element, or it can contain any number of elements. We will learn soon that it even can contain an **infinite** number of elements!

`Sequence` is a **generic class**. When writing the type of some sequence, we have to write `Sequence<T>`, where `T` stands for the type of element the sequence contains.

Write the type of the following:

Type in Java	**Description of Type**
`Sequence<Integer>`	A sequence of integer numbers
	A sequence of Strings
	A sequence of floating-point numbers
	A sequence of Boolean values
	A sequence of colors
	A sequence of graphics

The `T` above is the **type parameter** of the generic class. It can be replaced with almost any type. However, unfortunately, in Java, type parameters cannot be set to primitive types. `Sequence<int>` is not a legal type in Java.

Can we have a sequence containing the following type of elements?

- ☐ `double`
- ☐ `Character`
- ☐ `Boolean`
- ☐ `String`
- ☐ `Color`
- ☐ `Graphic`

Write the signature of a method named `colorDots` that takes a sequence of colors and returns a sequence of graphics. There’s no need to implement the method. We will do that later. For each color in the sequence, the method might produce a circle in that color.

```
public static
```

Write the signature of a method named `besides` that takes a sequence of graphics and returns a graphic. There’s no need to implement the method. We will do that later. The method will combine the graphics in the sequence besides each other into a single graphic.

```
public static
```

Constructing Sequences Library

A value of type `Sequence` is either an empty cell, or it is a cons cell containing an element and another value of type `Sequence`.

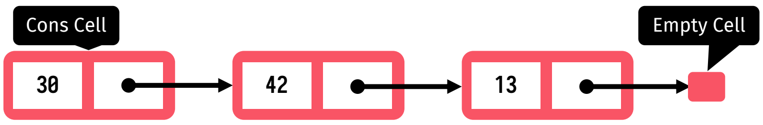


Diagram of the sequence of integers 30, 42, 13.

These two methods produce a sequence of something:

```
<T> Sequence<T> empty()

<T> Sequence<T> cons(T first, Sequence<T> rest)
```

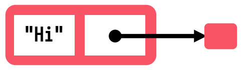
The `<T>` at the beginning specifies that for the given method, `T` is a type parameter. This means the method will produce a sequence of elements of type `T`, for any type `T`.

Here is an expression (a method invocation) that produces an empty sequence, a sequence containing no elements, a sequence of length 0:

```
empty()
```

Here is an expression that produces a `Sequence<String>` with one element, the `String` value `"Hi"`. This sequence has length 1.

```
cons("Hi", empty())
```



Write expressions that produce the following sequences. Draw an object diagram for each sequence.

The two elements `"Up"` and `"Down"`, such that the first element is `"Up"`, and the second element is `"Down"`:

The two floating-point numbers 0.0 and 0.1, in that order:

The colors `RED`, `GREEN`, and `BLUE`, in that order:

Deconstructing Sequences Library

Being able to “construct” a sequence is pointless if you cannot **do** anything with it! But ***what*** can you do with a sequence? For one, you can “deconstruct” it!

While the method `cons` constructs a cons cell from a **first element** and a **rest**, the methods `first` and `rest` deconstruct a cons cell into a **first element** and a **rest**.

```
<T> T first(Sequence<T> sequence)

<T> Sequence<T> rest(Sequence<T> sequence)
```

What happens if we try to deconstruct an empty sequence? If the sequence we want to deconstruct is not a cons cell, then we have a problem! We cannot deconstruct the empty cell! Thus, we need a way to tell whether a sequence is empty:

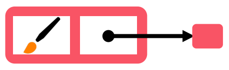
```
<T> boolean isEmpty(Sequence<T> sequence)
```

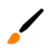
If `isEmpty` returns `true`, we know we cannot deconstruct the given sequence: we cannot call `first` and we cannot call `rest` on that sequence, because there is no cons cell to deconstruct.

Before we deconstruct sequences, let’s first build a few sequences. Let’s assume we have methods that provide Bill’s, Melinda’s, and Elon’s favorite colors:

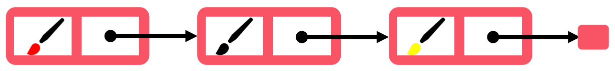
```
public static Sequence<Color> billFavs() {
  return cons(RED, cons(BLACK, cons(YELLOW, empty())));
}

public static Sequence<Color> melindaFavs() {
  return cons(Toolbelt.orange(), empty());
}

public static Sequence<Color> elonFavs() {
  return empty();
}
```

Let’s draw diagrams for the sequences the following expressions evaluate to:

`melindaFavs()` 

`first(melindaFavs())` 

`rest(melindaFavs())`

`billFavs()` 

`rest(billFavs())`

`first(rest(billFavs()))`

`rest(rest(billFavs()))`

`first(elonFavs())`

Are there any problems in evaluating the above expressions?

Carefully Deconstructing Language

Let's write a method that tells us whether a sequence of colors contains at least one color. If the sequence is not empty, it contains **at least one** color:

```
public static boolean atLeastOneColor(Sequence<Color> colors) {
  return isEmpty(colors) ? false : true;
}
```

The implementation of the above method can be simplified. Rewrite it:

```
  return
```

We can use the `atLeastOneColor` method to test whether someone has at least one favorite color. What do the following expressions evaluate to?

`atLeastOneColor( billFavs() )`	
`atLeastOneColor( melindaFavs() )`	
`atLeastOneColor( elonFavs() )`	

Write a method that returns `true` if the given sequence contains **at least two** colors, and that returns `false` otherwise. Your method is only allowed to call the `isEmpty`, `first`, and `rest` methods (remember, we can deconstruct a sequence using `isEmpty`, `first`, and `rest`), and it must use two conditional operators.

```
public static boolean atLeastTwoColors(Sequence<Color> colors) {

  return



}
```

The above code contained nested conditional operators, which become difficult to read and understand. Let's rewrite it. This time you can only use one call to each of `isEmpty` and `rest`. But you are allowed to call the `atLeastOneColor()` method!

```
public static boolean atLeastTwoColors(Sequence<Color> colors) {

  return

}
```

Values as Diagrams, Expressions as Trees Language

Write an expression that creates a sequence of the three characters 'A', 'B', and 'C':

Draw the **sequence** produced by this expression as a **diagram**:	Draw the **expression** itself as a **tree**:

Constructing Sequences: A More Convenient Way Library

To create a sequence, we have to write an expression of nested cons calls:

```
cons(1, cons(2, cons(3, cons(4, empty()))))
```

This is painful and verbose.
The `of` method provides a shortcut:

```
of(1, 2, 3, 4)
```

Draw the `of` expression as a tree:

Draw the `cons` expression as a tree:

The above two approaches produce the exact same sequence!

```
<T> Sequence<T> of(T element, …)
```

Note that the `of` method is very special. We have not seen a method like it before: It supports a variable number of arguments (zero or more).

Write expressions that use the method `of` to produce a sequence with:

The two elements `"Up"` and `"Down"`, such that the first element is `"Up"`, and the second element is `"Down"`:

The two floating-point numbers 0.0 and 0.1, in that order:

The boolean values `true`, `true`, `false`, and `true`, in that order:

The strings `"S"`, `"M"`, `"L"`, `"XL"`, in that order:

The colors `RED`, `GREEN`, and `BLUE`, in that order:

A red circle of diameter 40 and a green square of side 40, in that order:

Constructing Sequences: Implicitly Library

We do not need to explicitly enumerate each value of a sequence. JTamaro provides some methods that can create a complete sequence with a single call.

Replicating a value

The `replicate` method produces a sequence containing the same value a given number of times:

```
<T> Sequence<T> replicate(T element, int count)
```

```
replicate("Ciao.", 5)
```

...produces the same sequence as...

```
of("Ciao.", "Ciao.", "Ciao.", "Ciao.", "Ciao.")
```

Now, use `replicate` and whatever else is needed to produce:

```
of(10, 10, 10, 10, 10, 10)
```

```
of(RED, RED, RED, RED)
```

```
of(false, true, true, true, true, true)
```

The last task was special. How did you do it?

Creating a range of values

The `range` method works like the one you may have seen in Python:

```
Sequence<Integer> range(int start, int end, int step)
```

Complete the following table. Each row needs to contain two expressions that produce the same value:

With `range`:	**With** `of`:
`range(5)`	`of(0, 1, 2, 3, 4)`
`range(1, 3)`	`of(1, 2)`
`range(0, 10, 2)`	`of(0, 2, 4, 6, 8)`
`range(10, 0, -2)`	`of(10, 8, 6, 4, 2)`
	`of(1, 5, 9)`
`range(0, 5)`	
`range(0, 5, 1)`	
	`of(7, 3, -1)`
`range(1, 3, 0)`	
	`of()`

In some cases, you can express the same `of`-expression with different `range`-expressions. Also, some of the above calls to `range` make no sense.