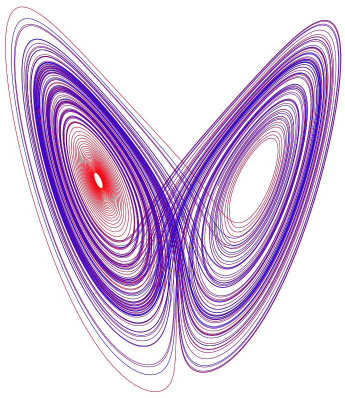
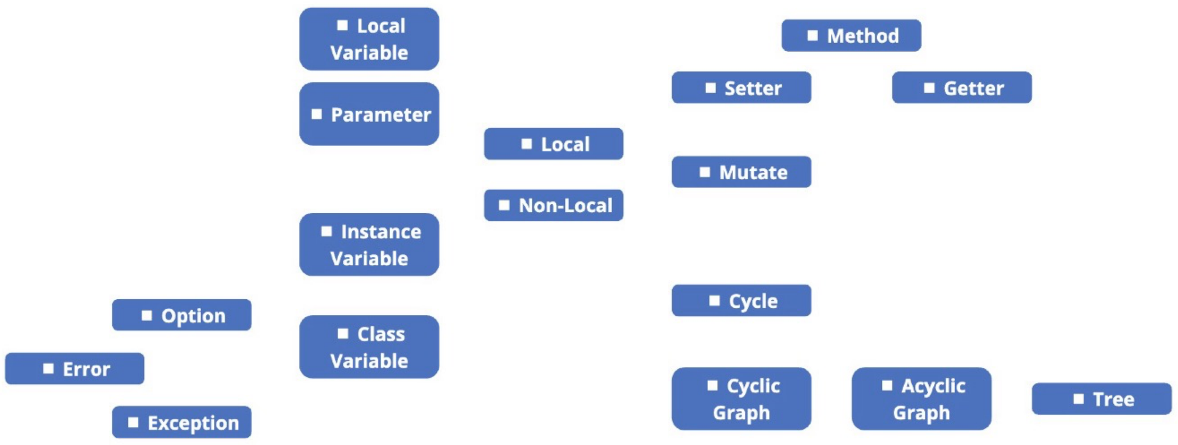
Non-Local Mutation

Student name:	TA signature:



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



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

`Option • None • Some • none • some • fold`

The special variable `this`

We already encountered the special variable `this`.

Which of the following claims are correct?

☐	`this` exists in every instance method
☐	`this` exists in every constructor
☐	`this` exists in every static method
☐	`this` holds a reference to the object the method or constructor operates on
☐	`this` gets initialized when the method or constructor is called

When there is no ambiguity, `this` can be left out. **Rewrite** class C on the right, the sweeter, "sugared" version of the code, leaving out `this` wherever possible:

Desugared

```
public class C {
  private final int x;
  private final int y;

  public C(int x, int initialY) {
    this.x = x;
    this.y = initialY;
  }

  public int x() {
    return this.x;
  }

  public int y() {
    return this.y;
  }

  public Pair<C,C> befriend(C o) {
    return new Pair<C,C>(o, this);
  }

  public Pair<C,C> meAndMyself() {
    return this.befriend(this);
  }

  public boolean same(C other) {
    return this == other;
  }
}
```

Sugared (short and sweet):

```
public class C {
  private final int x;
  private final int y;

  public C(int x, int initialY) {


  }

  public int x() {

  }

  public int y() {

  }

  public Pair<C,C> befriend(C o) {

  }

  public Pair<C,C> meAndMyself() {

  }

  public boolean same(C other) {

  }
}
```

Draw the memory diagram of **`new`** `C(1,2).meAndMySelf();` just before method `befriend` returns. The stack starts with `meAndMyself`.

Getter methods

When we define a **record** class, we provide its **components**:

```
public record Person(String name, int age) { }
```

Java automatically desugars this class, adding an **instance variable** and a getter method for each **component**.

```
public class Person {
  private final int age;
  …
  public int age() {
    return age;
  }
  …
}
```

If we have a `Person` object, and we want to access its `age`, how can we do that? Let's assume three scenarios:

Outside class Person	**Inside class Person, on different object**	**Inside class Person, on same object**

`public class Demo {`
 	
`  public static int m(`
`    Person p`
`  ) {`
 	
`    return p.age();`
`  }`
`}`
	`public class Person {`
	`  …`
	`  public int m(`
	`    Person p`
	`  ) {`
	`    assert p != this;`
	`    return p.age();`
	`  }`
	`}`
		`public class Person {`
		`  …`
		`  public int m() {`
		
		`    return this.age();`
		`  }`
		`}`

Check in which scenarios we can legally replace the accessor method call with an instance variable access?

☐ `return p.age;`	☐ `return p.age;`	☐ `return this.age;`

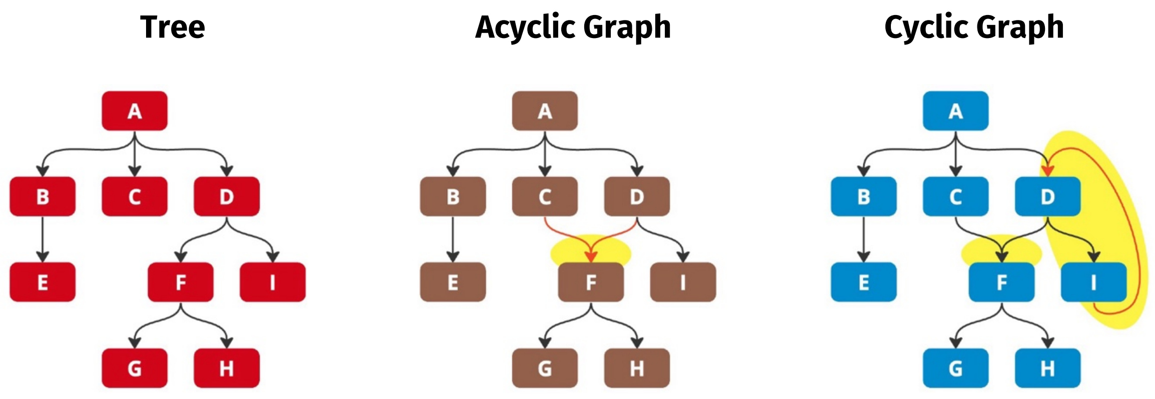
In the last example, could we leave out the `this`? In both cases? Briefly **explain**!

Getter methods are **automatically** generated when we define record components. If we create classes that are not records, we can **manually** write getter methods. Usually, we use the name `getXxx()` instead of just `xxx()`, for a getter of field `xxx`.

Note that it is not necessary to have getter methods. Often it is not even a good idea! If all your classes have getter methods for all their fields, then your code might **not** be very well designed.

On Trees, Acyclic Graphs, and Cyclic Graphs

There are some fundamental data structures you want to use in your programs.



Trees: No Sharing, No Cycles

We can build a tree by defining a record class and using a single expression to construct the tree:

```
public interface Tree {}
public record Node(char label, Sequence<Tree> children) implements Tree { }
```

Write an expression that uses the given types to create the red tree above:

```
new
```

Acyclic Graphs: Sharing, No Cycles

We can build an acyclic graph by defining a record class and using a way to name a substructure so we can refer to that substructure multiple times (sharing):

```
public interface DAG {}
public record Node(char label, Sequence<DAG> children) implements DAG { }
```

Write the code that uses the given types to create the brown acyclic graph above:

Can you do so in a single expression? Why, or why not?

LüCE Lugano Computing Education Research Lab

Cyclic Graphs: Sharing, Cycles

In addition to **sharing** (two variables pointing to the same object), a **cyclic graph** also allows **cycles**.

Try to build a cyclic graph **using record classes** in a similar way to how we did it above for trees and acyclic graphs:

```
public interface Graph {}

public record Node(
```

Given your classes, try to **write** the code that uses the given types to create the blue cyclic graph above:

We seem to have a chicken-and-egg problem here. If we want to create a `Node` object D that refers to another `Node` object I, we first need I. But if we want to create I, we first need D, because I refers to D. We have a **cycle**.

If we want to create cyclic object structures, we cannot do so in one go. We have to first create the objects, and only then set them up so they point to each other. To establish the **cycle**, we might need to **mutate** at least one of the involved objects.

Local vs. Non-Local Mutation

Mutating an object means mutating one of its instance variables. Unlike the mutation of a **local variable** or a **parameter**, which is only observable within a specific method, the mutation of an **instance variable** could be observable anywhere in the program, even if the variable is private. **Class variables** are similarly problematic.

If we mutate a local variable or parameter, we say this is **local mutation**.

If we mutate an instance variable or a class variable, we say this is **non-local mutation**.

Simple cyclic structures

Let's look at some simple, minimalistic cyclic structures. Once we understand how they work, we will then see more realistic cases.

The simplest cyclic structure is an object referring to itself (a self-loop):

```java
public class It {

  private It other;

  public It() { }

  public void setOther(It other) {
    this.other = other;
  }
}

It o = new It();
o.setOther(o);
```

A cycle can be bigger, though! Two objects of the same class forming a cycle:

```java
It a = new It();
It b = new It();
a.setOther(b);
b.setOther(a);
```

Cycles can involve objects of different types:

```java
public class Ping {
  private Pong pong;
  public Pong() { }
  public void setPong(Pong pong) { this.pong = pong; }
}
public class Pong {
  private Ping ping;
  public Ping() { }
  public void setPing(Ping ping) { this.ping = ping; }
}

Ping pi = new Ping();
Pong po = new Pong();
pi.setPong(po);
po.setPing(pi);
```

Note that in all the above cases, we **mutate** an instance variable (with an assignment `=`) **after** the object has been created. We provide a **setter method** (`setOther`, `setPong`, `setPing`) that performs that mutation for us.

Another Example of Cyclic Data: Your Social Network

Except for the previous page, we have not yet had a need for creating cyclic data structures. But in the real world, cyclic structures are prevalent. One example: your network of friends!

```
public class Person {

  private Sequence<Person> friends;

  public Person() {
    friends = empty();
  }

  …

}
```

Friendships usually are bidirectional: if A is my friend, I am A's friend. It's a cycle!

Let's write a method that allows us to establish such a connection between two persons:

```
public static void formFriendship(Person a, Person b) {
  a.friends = cons(b, a.friends);
  b.friends = cons(a, b.friends);
}
```

For this method to work, we had to mutate an instance variable! Specifically, we had to change the value in `a.friends` (in the first assignment). On top of this, we also changed the value in `b.friends` (in the second assignment).

We don't necessarily have to establish only bidirectional friendships. We also could allow the creation of unidirectional "friendships": I might like A, but A might not like me:

```
public void addFriend(Person a) {
  this.friends = cons(a, this.friends);
}
```

We still could establish a cycle:

```
Person x = new Person();
Person y = new Person();
x.addFriend(y);
y.addFriend(x);
```

But we also could establish a unidirectional friendship:

```
Person orsino = new Person();
Person olivia = new Person();
orsino.addFriend(olivia);
```

The Risks of Non-Local Mutation (Mutation of Fields)

If a method mutates a field, this change can be observable **after** that method returns. This can be very risky.

Example: A mutable Date

Assume a mutable `Date` class. The mutator method, `later`, allows you to move the date by the given number of days:

```
public class Date {
  private int unixDay; // days since 1 January 1970
  public Date(int year, int month, int day) { … }
  public void later(int days) { unixDay = unixDay + days; }
  public String toString() { … }
}
```

Now assume you are planning your graduation. You already know the date of your graduation, and you want to organize a party 2 days later:

```
public class Plans {
  public static void organize(Date graduation) {
    masterPlan(graduation, party(graduation));
  }
  private static Date party(Date graduation) {
    Date party = graduation;
    party.later(2);
    return party;
  }
  private static void masterPlan(Date graduation, Date party) {
    // what is the program state at this point?
    …
  }
}
```

Draw the **Memory Diagram** at the marked point in the execution of this program, assuming the program starts with the call `Plans.organize(new Date(1970, 1, 11))`. Assume that the constructor of `Date` will set the field `unixDay` to 10. Don't draw the stack frame for the constructor of `Date` and erase the frames of the methods that returned. Is there anything unexpected happening?

Updating our computational model

We have been using various diagrams (notional machines) to **reason** about different aspects of programs. We can consider them effectively as our computational model of Java. As we introduced more features of Java we had to introduce new diagrams to help us reason about those features. For example, when we introduced statements (e.g., loops), we had to introduce the *Control-Flow Graph* to reason about control-flow over multiple statements in a method. We can then use them to reason about programs in other languages too, so about programming in general!

What diagrams have we seen? Summarize, for each diagram, the aspects of programming they focus on.

Sometimes, new features required us to extend diagrams that were shown before in simpler forms. For example, in Workbook 9 we introduced mutation of local variables and we had to extend the *Memory Diagram* to show local variables (in the Stack). In addition to that, we showed how every variable (including fields) has always a name, a type, and a value. At that point we showed how, in Java, variables of primitive types (e.g., `int` or `double`) directly contain the values themselves but variables of reference types (e.g., `String`) contain references to objects.
We represented a reference as an arrow pointing to the corresponding object or with an @ followed by a number (the object's address). The result of evaluating an expression that produces an object is not "the object" but a reference to it. Understanding that distinction is essential when we **mutate fields**! Therefore, it's essential to extend **all the diagrams** that show values to represent the objects with @ followed by a number.

Draw an **Expression Tree** for `masterPlan(graduation, party(graduation))`.
Annotate each node with its **type**. Annotate each node with its **value**, assuming the same program execution illustrated in the memory diagram on the previous page.

Draw the **Dynamic Call Tree** for `Plans.organize(new Date(1970, 1, 11))`.
Using again the context we used for the memory diagram, show all **argument values** for each call of a method with parameters, and indicate the **return value** of each call right below the call.

Another example: A mutable Hand in a game

The method on the left has a lot of code duplication. Your teammate refactored it:

Original code	**Your teammate's refactored code**

```
public class Controller {
  public Game buildGame() {
    Game initialGame = new Game();
    initialGame.addHand(new Hand(7));
    initialGame.addHand(new Hand(7));
    initialGame.addHand(new Hand(7));
    initialGame.addHand(new Hand(7));
    initialGame.addHand(new Hand(7));
    return initialGame;
  }
}
```
	```
	public class Controller {
	  public Game buildGame() {
	    Game initialGame = new Game();
	    Hand hand = new Hand(7);
	    for (int i = 0; i < 5; i++) {
	      initialGame.addHand(hand);
	    }
	    return initialGame;
	  }
	}
	```

If `Hand` is mutable, is the refactored code equivalent to the original code? Why?

Sketch an implementation of (1) a mutable `Hand` class with a method `draw` that adds a card to a hand, (2) the class `Game` with a method `addHand`, and (3) a fix in your teammate's refactoring of the method `buildGame`.

Handling Errors

While programming, our functions may have to deal with parameter values for which they cannot produce a meaningful result. We could handle this with assert statements, but they would terminate the program. If we don't want to terminate in such a situation, we need a way to deal with these "errors". For example, take a look at this method to compute the average of a sequence of numbers:

```
double average(Sequence<Double> nums) {
  return sum(nums) / length(nums);
}
```

What can go wrong?

A non-solution: return a special value

In some cases, programmers try to fix this situation by returning a special, distinguished value (also dubbed a *sentinel*). For example, consider a method

```
<T> int indexOf(T element, Sequence<T> values)
```

that returns the position of a given element in a sequence. When the element is not found, it is customary to return `-1` as a special value.

While this is convenient, it is also **extremely** error prone. All the pieces of code that call `indexOf` need to remember that there is a special value that needs to be handled. Programmers are humans, they forget to deal with the special case, and then the program inevitably crashes.

Even worse, returning a special value is **NOT** a solution. Which special value would you return in the above `average` method when `nums` is an empty sequence to avoid performing a division by zero?

No value of type `double` is a good one, negatives included! Returning `-1.0` as it is done in `indexOf` does not work, because `Sequence.of(-2, 0)` has exactly average `-1`.

Java's historical solution: exceptions

Historically, Java programmers extensively use exceptions to signal problematic cases (such as the ones above). You have already probably witnessed exceptions in the wild while programming: did you get a `NullPointerException`? Or an `IndexOutOfBoundsException`?

These exceptions, however, only manifest at **runtime**, while the program is running. Can we do better? And **prevent** all these situations **statically**?

The Option type

There is an elegant idea that addresses this kind of situation and is increasingly getting adopted in modern programming languages: the idea of an **Option**.

The idea is the following: instead of promising to always return a value of type `T`, we specify as a return type **`Option<T>`**.

We cannot instantiate an `Option<T>`. We can **either** have a value of the type **`Some<T>`** or a value of type **`None<T>`**.

JTamaro provides a static factory method `Options.none()` to create a `None<T>` value. Correspondingly, there is a static factory method `Options.some(T)` that creates `Some<T>` values starting from values of type `T`.

Refactor the implementation of `average` to return values of the `Option` type:

```
Option<Double> average(Sequence<Double> nums) {
  return
          ?
          :
}
```

Once we get a value of type `Option<T>`, what can we do with it?

We cannot extract the value of type `T` we would like, because there might not be one! Instead, we are forced to **handle both cases**.

An object of class `Option<T>` has the following instance method:

```
<R> R fold(Function1<T, R> someCase, R noneValue);
```

When Option contains **some** value of type `T`, the function `someCase` is executed to produce a value (of a possibly different type `R`).
When it contains **none**, `fold` just returns the provided `noneValue` (also of type `R`).

Use `fold` to produce a string representation of a `Student`, such that it returns strings like "`Luca – GPA: 5.5`" and "`Igor – No grades yet`" depending on the situation.

```
record Student(String name, Sequence<Double> grades) {
  public String toString() {
    return name + " – " +



  }
}
```