1. Problem 1: Message Processing Semantics

Consider the following actor system:

```
enum Protocol:
  case Write(value: Int)
  case Read(requester: ActorRef)
import Protocol.*

enum Responses:
  case Answer(value: Int)
import Responses.*

class Memory extends Actor:
  var value = 0

  override def receive: Receive = {
    case Write(newValue) => value = newValue
    case Read(requester) => requester ! Answer(value)
  }

class Client(memory: ActorRef) extends Actor:
  override def receive: Receive = { case Answer(value) =>
    println(value)
  }

class MyProxy(memory: ActorRef) extends Actor:
  override def receive: Receive = { case message =>
    memory ! message
  }
```

1.1. Problem 1.1

And the following test:

```
@main def problem1_1 =
  for _ <- 1 to 1000 do
    val system = ActorSystem("example")
    try
      val memory = system.actorOf(Props(Memory()))
      val client = system.actorOf(Props(Client(memory)))
      memory ! Read(client)
```

```
      memory ! Write(1)
    finally system.terminate()
```

What are the possible values printed by the `println` command in the `Client` actor? Why?

1.2. Problem 1.2

Now, consider the following test:

```
@main def problem1_2 =
  for _ <- 1 to 1000 do
    val system = ActorSystem("example")
    try
      val memory = system.actorOf(Props(Memory()))
      val proxy  = system.actorOf(Props(MyProxy(memory)))
      val client = system.actorOf(Props(Client(memory)))
      proxy ! Read(client)
      memory ! Write(1)
    finally system.terminate()
```

1. What are the possible values printed by the `println` command in the `Client2` actor? Why?
2. Would the output be different if the `Read` and `Write` messages were issued in the other order?
3. What if both messages are sent through the `Proxy` actor?

2. Problem 2: The Josephus Problem

In this exercise, we will revisit the famous *Josephus problem*. In this problem, a group of soldiers trapped by the enemy decide to commit suicide instead of surrendering. In order not to have to take their own lives, the soldiers devise a plan. All soldiers are arranged in a single circle. Each soldier, when it is their turn to act, has to kill the next soldier alive next to them in the clockwise direction. Then, the next soldier that is still alive in the same direction acts. This continues until there remains only one soldier in the circle. This last soldier is the lucky one, and can surrender if he decides to. The *Josephus problem* consists in finding the position in the circle of this lucky soldier, depending on the number of soldiers.

In this exercise, you will implement a *simulation* of the mass killing of the soldiers. Each soldier will be modeled by an actor. Soldiers are arranged in a circle and when their turn comes to act, they kill the next alive soldier in the circle. The next soldier that is still alive in the circle should act next. The last soldier remaining alive does not kill himself but prints out its number to the standard output.

The code below covers the creation of all actors and the initialization of the system. Your goal is to implement the `receive` method of `Soldier`.

Hint: Think about what state the soldier must have.

```
import akka.actor.*

object Soldier:
  // The different messages that can be sent between the actors:
  enum Protocol:

    // The recipient should die.
    case Death

    // The recipient should update its next reference.
    case Next(next: ActorRef)

    // The recipient should act.
    case Act

class Soldier(number: Int) extends Actor:
  import Soldier.*
```

```
  import Protocol.*

  def receive: Receive = behavior(None, None, false)

  def behavior(
      next: Option[ActorRef],
      killer: Option[ActorRef],
      mustAct: Boolean
  ): Receive = ???

@main def problem2(n: Int) =
  import Soldier.*
  import Soldier.Protocol.*

  // Initialization
  val system = ActorSystem("mySystem")
  require(n >= 1)

  // Creation of the actors.
  val actors = Seq.tabulate(n)(
    (i: Int) => system.actorOf(Props(classOf[Soldier], i), "Soldier"
+ i)
  )

  // Inform all actors of the next actor in the circle.
  for i <- 0 to (n - 2) do actors(i) ! Next(actors(i + 1))
  actors(n - 1) ! Next(actors(0))

  // Inform the first actor to start acting.
  actors(0) ! Act
```