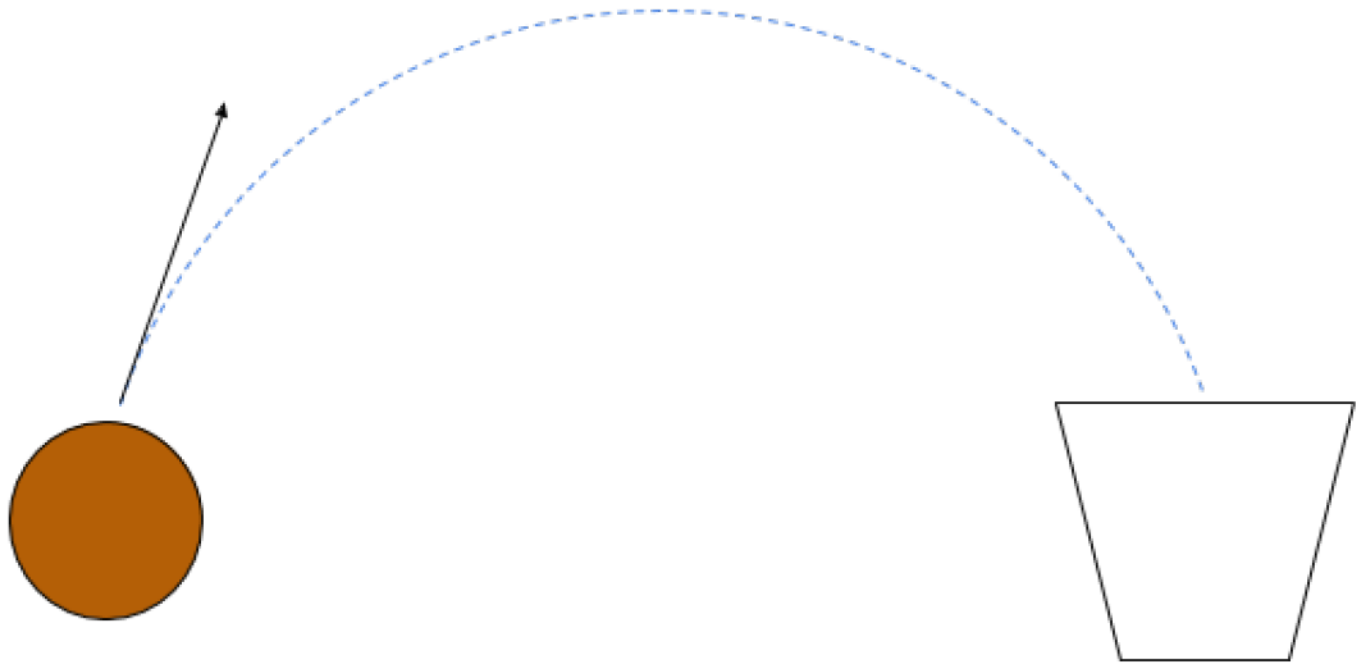
More Loop Examples

Effective Programming in Scala

Another Example: Basketball

Given the position of a basketball player, and a shot direction and force, is the ball going to pass through the hoop?



Basketball

```
case class Position(x: Double, y: Double):
  def distanceTo(that: Position): Double = ???
  def distanceToLine(line: (Position, Position)): Double = ???

object Position:
  val player = Position(0, 1.80)
  val hoop   = Position(6.75, 3.048)

case class Angle(radians: Double)
case class Speed(metersPerSecond: Double)

def isWinningShot(angle: Angle, speed: Speed): Boolean
```

Implementation Sketch

- Compute the successive positions of the ball over time,
- The algorithm terminates in two cases:
  - If the ball passes through the hoop, this is a winning shot, true
  - If the ball ends up too far (on the ground or beyond the hoop), this is a failure. False

To check whether the ball passed through the hoop, we compute the distance between the center of the hoop and the straight segment made of two successive positions.

Prelude

```
def isWinningShot(angle: Angle, speed: Speed): Boolean =
  val v0X = speed.metersPerSecond * math.cos(angle.radians)
  val v0Y = speed.metersPerSecond * math.sin(angle.radians)
  val p0X = Position.player.x
  val p0Y = Position.player.y
  val g = -9.81
  ...
```

Utility Methods

```
def isWinningShot(angle: Angle, speed: Speed): Boolean =
  ...
  def goesThroughHoop(line: (Position, Position)): Boolean =
    Position.hoop.distanceToLine(line) < 0.01
  def isNotTooFar(position: Position): Boolean =
    position.y > 0 && position.x <= Position.hoop.x + 0.01
  def position(t: Int): Position =
    val x = p0X + v0X * t
    val y = p0Y + v0Y * t + 0.5 * g * t * t
    Position(x, y)
  ...
```

Implementation Using Collections

We will use a type of collection named `LazyList`. You can think of them as lists whose elements are computed only when needed.

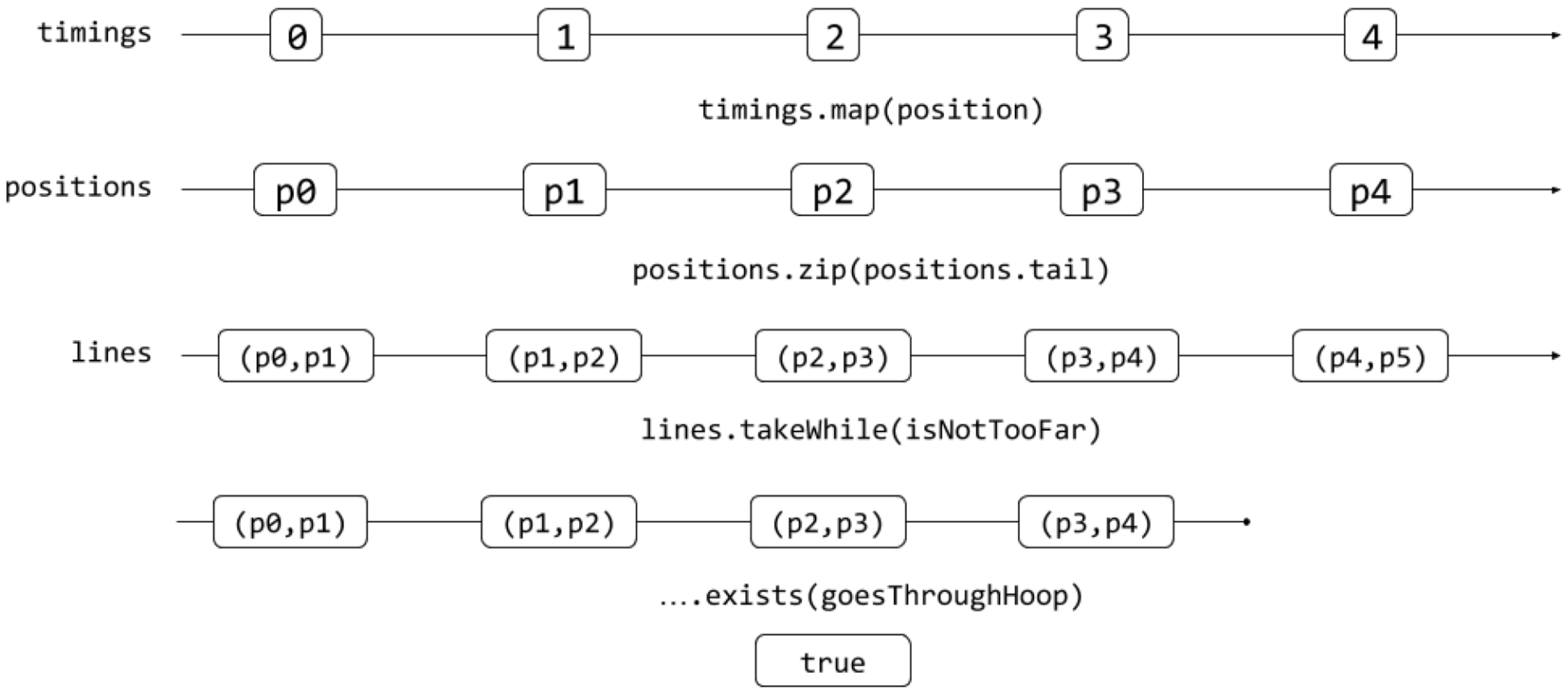
This allows us to model infinite collections.

For instance, `LazyList.from(0)` constructs a lazy list with the elements 0, 1, 2, etc.

Implementation Using Collections (2)

```
def isWinningShot(angle: Angle, speed: Speed): Boolean =
  ...
  val timings: LazyList[Int] = LazyList.from(0)
  val positions: LazyList[Position] =
    timings.map(position)
  val lines = positions.zip(positions.tail)
  lines
    .takeWhile((p1, _) => isNotTooFar(p1))
    .exists(goesThroughHoop)
```

Implementation Using Collections (3)



Implementation Using `while`

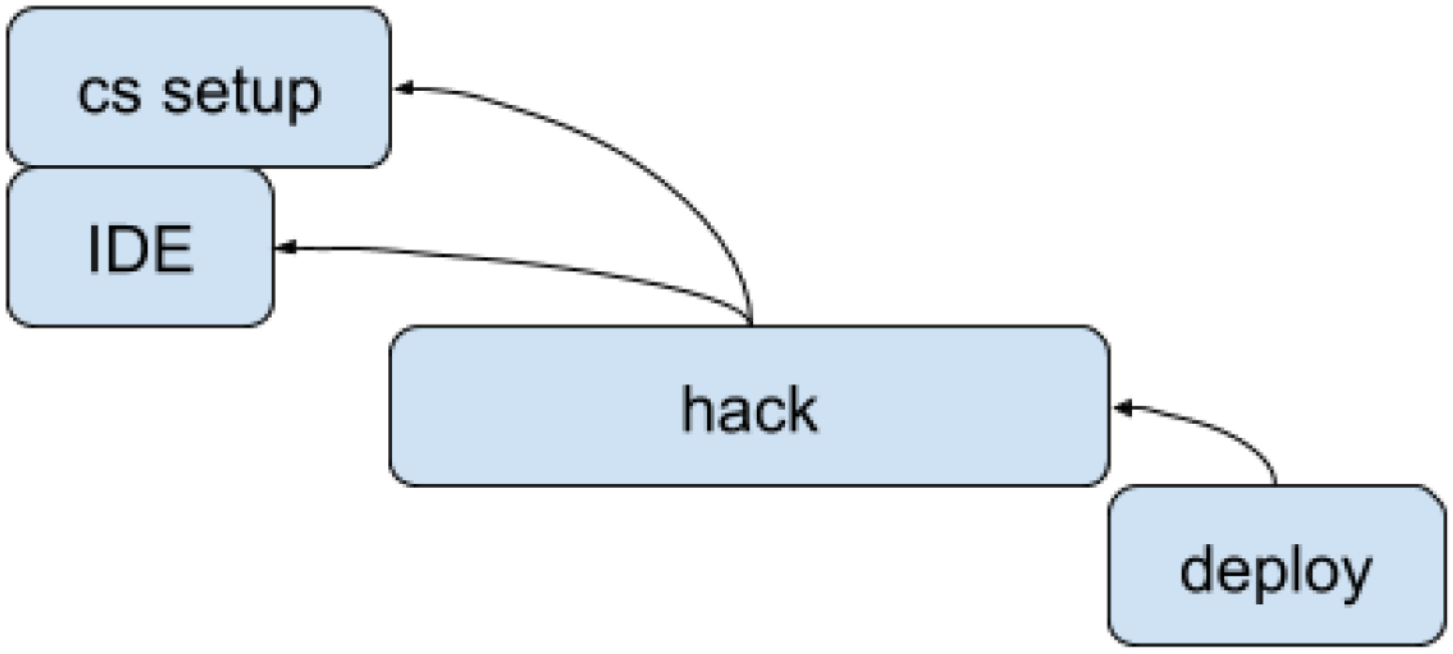
```
def isWinningShot(angle: Angle, speed: Speed): Boolean =
  ...
  var previousPosition = position(0)
  var isWinning = false
  var time = 1
  while isNotTooFar(previousPosition) && !isWinning do
    val nextPosition = position(time)
    val line = (previousPosition, nextPosition)
    isWinning = goesThroughHoop(line)
    time = time + 1
    previousPosition = nextPosition
  isWinning
```

Implementation Using Recursion

```
def isWinningShot(angle: Angle, speed: Speed): Boolean =
  ...
  def loop(time: Int): Boolean =
    val currentPosition = position(time)
    if isNotTooFar(currentPosition) then
      val nextPosition = position(time + 1)
      val line = (currentPosition, nextPosition)
      goesThroughHoop(line) || loop(time + 1)
    else
      false

  loop(time = 0)
```

Last Example: Task Management



Question: When can I expect the `deploy` task to be finished at best?

Model

```
case class Task(name: String, duration: Int, requirements: List[Task])
```

Example:

```
val csSetup = Task("cs setup", 4, Nil)
val ide     = Task("IDE",      3, Nil)
val hack    = Task("hack",     8, List(csSetup, ide))
val deploy  = Task("deploy",   3, List(hack))
```

Implementation Using Recursion

```scala
def maxTotalDuration(tasks: List[Task]): Int =
  tasks match
    case Nil          => 0
    case head :: tail =>
      val tailDuration = maxTotalDuration(tail)
      val headDuration = totalDuration(head)
      if headDuration < tailDuration then tailDuration
      else headDuration

def totalDuration(task: Task): Int =
  task.duration + maxTotalDuration(task.requirements)
```

Taking Advantage of `List` Operations

```
def totalDuration(task: Task): Int =
  val requirementsMaxTotalDuration =
    task.requirements
      .map(totalDuration)
      .maxOption
      .getOrElse(0)
  task.duration + requirementsMaxTotalDuration
```

Structural Recursion

```
def maxTotalDuration(tasks: List[Task]): Int =
  tasks match
    case Nil          => 0
    case head :: tail =>
      val tailDuration = maxTotalDuration(tail)
      val headDuration = totalDuration(head)
      ...
```

Note that `List` is a recursive type: a `tail` is itself a `List`.

We also recursively apply the method `maxTotalDuration` to the `tail` of the `List[Task]`.

This pattern is called **structural recursion**.

Generally, recursive data types are easy to process with structurally recursive algorithms.

Summary

Several styles of loops are possible.

Often, exploring the solution space of a problem is easy to express with the existing collections operations.

Generally, recursion works well with recursive data types.

According to the situation, pick the solution that works better for you and your team.