



# RECURSIONS

# Recursions

Recursive data structures

Recursive functions

# Recursive functions

```
def printMany(n: Int, message: String): Unit =  
  if(n <= 0) () // do nothing  
  else {  
    println(message)  
    printMany(n - 1, message)  
  }
```

```
printMany(3, "FP is awesome")  
// FP is awesome  
// FP is awesome  
// FP is awesome
```

# Recursive functions

```
def printMany(n: Int, message: String): Unit =  
  if(n <= 0) () // do nothing  
  else {  
    println(message)  
    printMany(n - 1, message)  
  }
```

```
printMany(3, "FP is awesome")  
// FP is awesome  
// FP is awesome  
// FP is awesome
```

```
def printMany(n: Int, message: String): Unit =  
{  
  var counter = n  
  while (counter > 0) {  
    counter -= 1  
    println("FP is awesome")  
  }  
}
```

```
printMany(3, "FP is awesome")  
// FP is awesome  
// FP is awesome  
// FP is awesome
```

# Recursive functions

```
def printMany(n: Int, message: String): Unit =  
  if(n > 0) {  
    println(message)  
    printMany(n - 1, message)  
  }  
  else ()
```

```
printMany(3, "FP is awesome")  
// FP is awesome  
// FP is awesome  
// FP is awesome
```

```
def printMany(n: Int, message: String): Unit =  
{  
  var counter = n  
  while (counter > 0) {  
    counter -= 1  
    println("FP is awesome")  
  }  
}
```

```
printMany(3, "FP is awesome")  
// FP is awesome  
// FP is awesome  
// FP is awesome
```

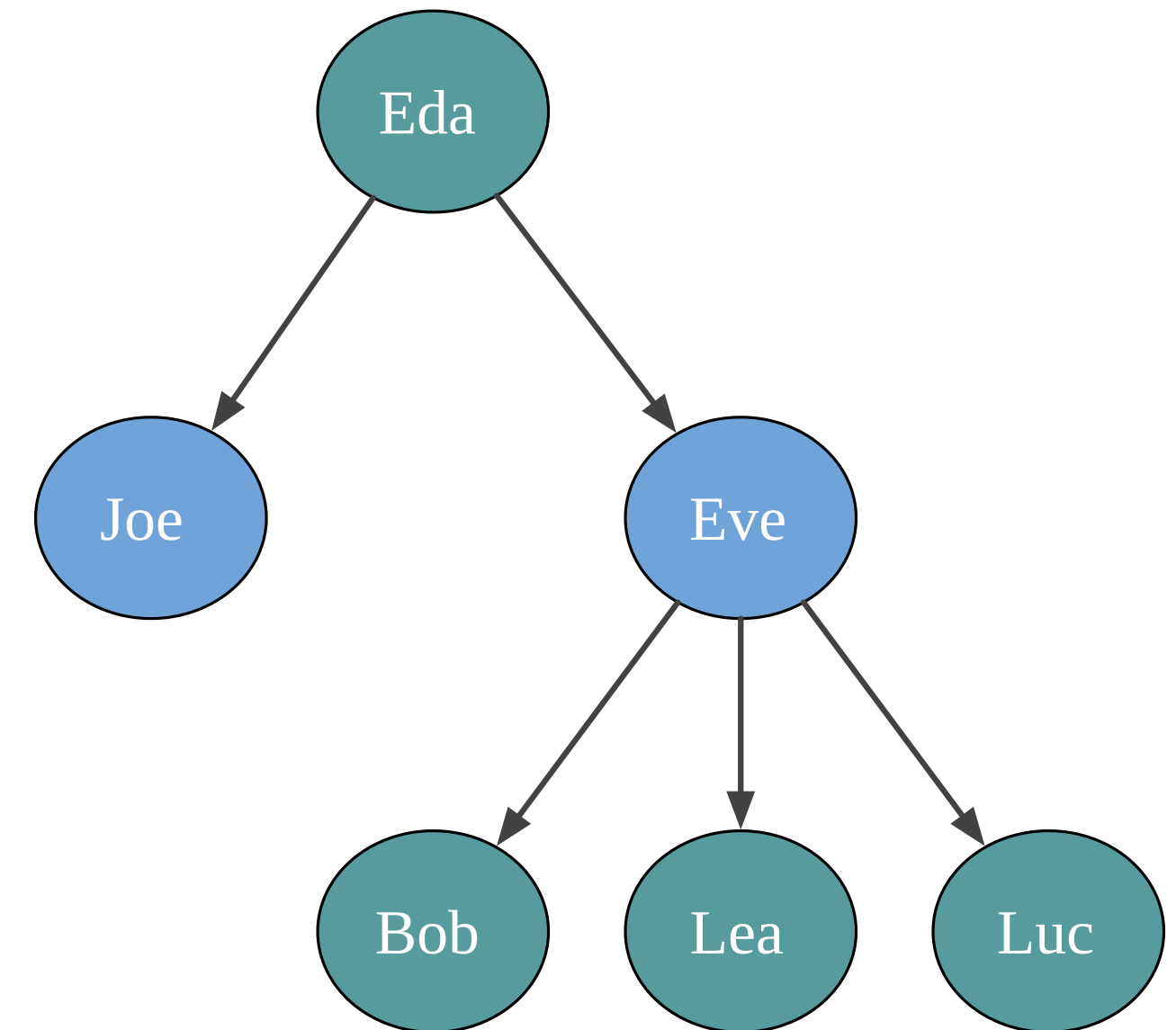
# Recursive functions

```
def printMany(n: Int, message: String): Unit =  
  (1 to n).foreach(_ => println(message))
```

```
printMany(3, "FP is awesome")  
// FP is awesome  
// FP is awesome  
// FP is awesome
```

# Recursive data structures

```
case class Person(name: String, children: List[Person])  
  
val bob = Person("Bob", Nil)  
val lea = Person("Lea", Nil)  
val luc = Person("Luc", Nil)  
  
val eve = Person("Eve", List(bob, lea, luc))  
val joe = Person("Joe", Nil)  
  
val eda = Person("Eda", List(joe, eve))
```



# Recursive data structures

## JSON

```
{  
  "name": "John Doe",  
  "age": 25,  
  "address": {  
    "street": {  
      "number": 12,  
      "name": "Cody road"  
    },  
    "country": "UK"  
  }  
}
```

## YAML

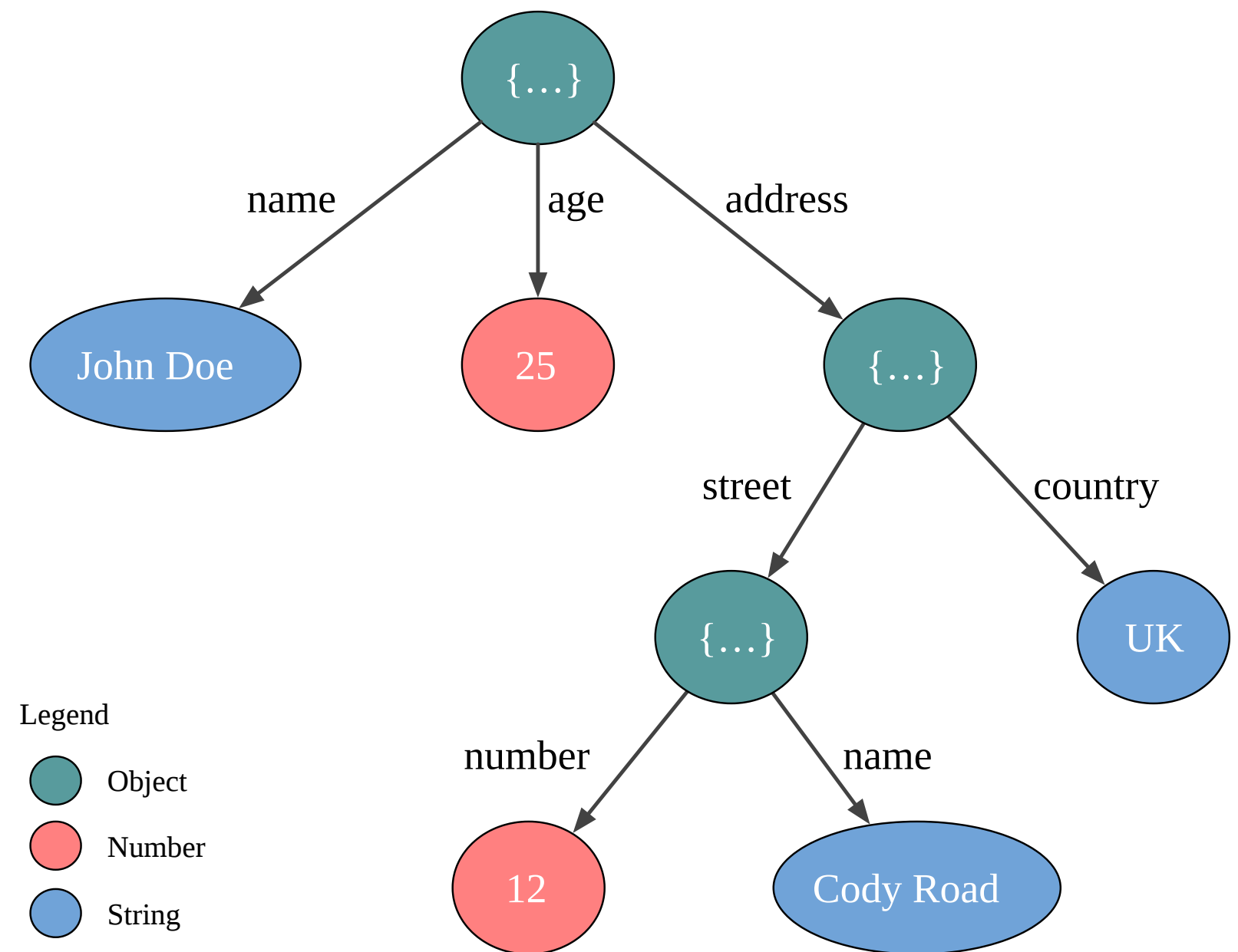
```
name: John Doe  
age: 25  
address:  
  street: 12  
    number: 12  
    name: Cody road  
country: UK
```



# Recursive data structures

## JSON

```
{  
  "name": "John Doe",  
  "age": 25,  
  "address": {  
    "street": {  
      "number": 12,  
      "name": "Cody road"  
    },  
    "country": "UK"  
  }  
}
```



# Is JSON a case class?

```
case class Json(  
  number: Double,  
  text   : String,  
  obj    : Map[String, Json],  
)
```

# Is JSON a case class?

```
case class Json(  
  number: Option[Double],  
  text   : Option[String],  
  obj    : Option[Map[String, Json]],  
)  
  
val json = Json(  
  number = None,  
  text   = Some("John Doe"),  
  obj    = None  
)
```

# Is JSON a case class?

```
case class Json(  
  number: Option[Double],  
  text   : Option[String],  
  obj    : Option[Map[String, Json]],  
)  
  
val json1 = Json(  
  Some(25),  
  Some("John Doe"),  
  None  
)  
  
val json2 = Json(None, None, None)
```

# JSON is a recursive enumeration

```
enum Json {  
  
  // Leaves  
  case JsonNumber(number: Double)  
  case JsonString(text  : String)  
  
  // Branch  
  case JsonObject(obj: Map[String, Json])  
  
}
```

# JSON is a recursive enumeration

```
enum Json {  
    // Leaves  
    case JsonNumber(number: Double)  
    case JsonString(text : String)  
  
    // Branch  
    case JsonObject(obj: Map[String, Json])  
}  
  
val json: Json = Json.JsonNumber(25)  
  
import Json._  
  
val number: Json = JsonNumber(25)  
val text  : Json = JsonString("John Doe")  
val obj   : Json = JsonObject(Map())
```

# JSON is a recursive enumeration

## In Scala 3

```
enum Json {  
  
  // Leaves  
  case JsonNumber(number: Double)  
  case JsonString(text : String)  
  
  // Branch  
  case JsonObject(obj: Map[String, Json])  
  
}
```

## In Scala 2

```
sealed trait Json  
  
// Leaves  
case class JsonNumber(number: Double) extends Json  
case class JsonString(text : String) extends Json  
  
// Branch  
case class JsonObject(obj: Map[String, Json])  
  extends Json
```

# JSON is a recursive enumeration

```
sealed trait Json
```

```
// Leaves
```

```
case class JsonNumber(number: Double) extends Json
```

```
case class JsonString(text : String) extends Json
```

```
// Branch
```

```
case class JsonObject(obj: Map[String, Json])  
  extends Json
```

```
val number = JsonNumber(12)
```

```
// number: JsonNumber = JsonNumber(12.0)
```

```
val json: Json = JsonNumber(12)
```

```
// json: Json = JsonNumber(12.0)
```



# Working with recursive enumerations

```
val john: Json = JsonObject(Map(  
  "name"      -> JsonString("John Doe"),  
  "age"       -> JsonNumber(25),  
  "email"     -> JsonString(" john@doe.com  "),  
  "address"   -> JsonObject(Map(  
    "street-number" -> JsonNumber(12),  
    "post-code"     -> JsonString("E16 4SR  ")  
  ))  
))
```

# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  ???
```

# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  json match {  
    case JsonNumber(num)   => ???  
    case JsonString(text)  => ???  
    case JsonObject(obj)   => ???  
  }
```

# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  json match {  
    case JsonNumber(num) => ???  
    case JsonObject(obj) => ???  
  }  
// warning: match may not be exhaustive.  
// It would fail on the following input: JsonString(_)  
//   json match {  
//     ^^^^
```

## Transform this warning into an error

```
scalacOptions += "-Wconf:cat=other-match-analysis:error"
```

# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  json match {  
    case JsonNumber(num) => JsonNumber(num) // do nothing  
    case JsonString(text) => ???  
    case JsonObject(obj)  => ???  
  }
```

# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  json match {  
    case _: JsonNumber    => json  
    case JsonString(text) => ???  
    case JsonObject(obj)  => ???  
  }
```

# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  json match {  
    case _: JsonNumber    => json  
    case JsonString(text) => JsonString(text.trim)  
    case JsonObject(obj)  => ???  
  }
```

# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  json match {  
    case _: JsonNumber    => json  
    case JsonString(text) => JsonString(text.trim)  
    case JsonObject(obj)  =>  
      val newObj = obj.map {  
        case (key, value) => (key, trimAll(value))  
      }  
      JsonObject(newObj)  
  }
```



# Working with recursive enumerations

```
def trimAll(json: Json): Json =  
  json match {  
    case _: JsonNumber    => json  
    case JsonString(text) => JsonString(text.trim)  
    case JsonObject(obj)  =>  
      val newObj = obj.map {  
        case (key, value) => (key, trimAll(value))  
      }  
      JsonObject(newObj)  
  }
```

```
john  
// john: Json = JsonObject(Map(  
//   "name"      -> JsonString("John Doe"),  
//   "age"       -> JsonNumber(25),  
//   "email"     -> JsonString(" john@doe.com  "),  
//   "address"  -> JsonObject(Map(  
//     "street-number" -> JsonNumber(12),  
//     "post-code"    -> JsonString("E16 4SR  ")  
//   ))
```

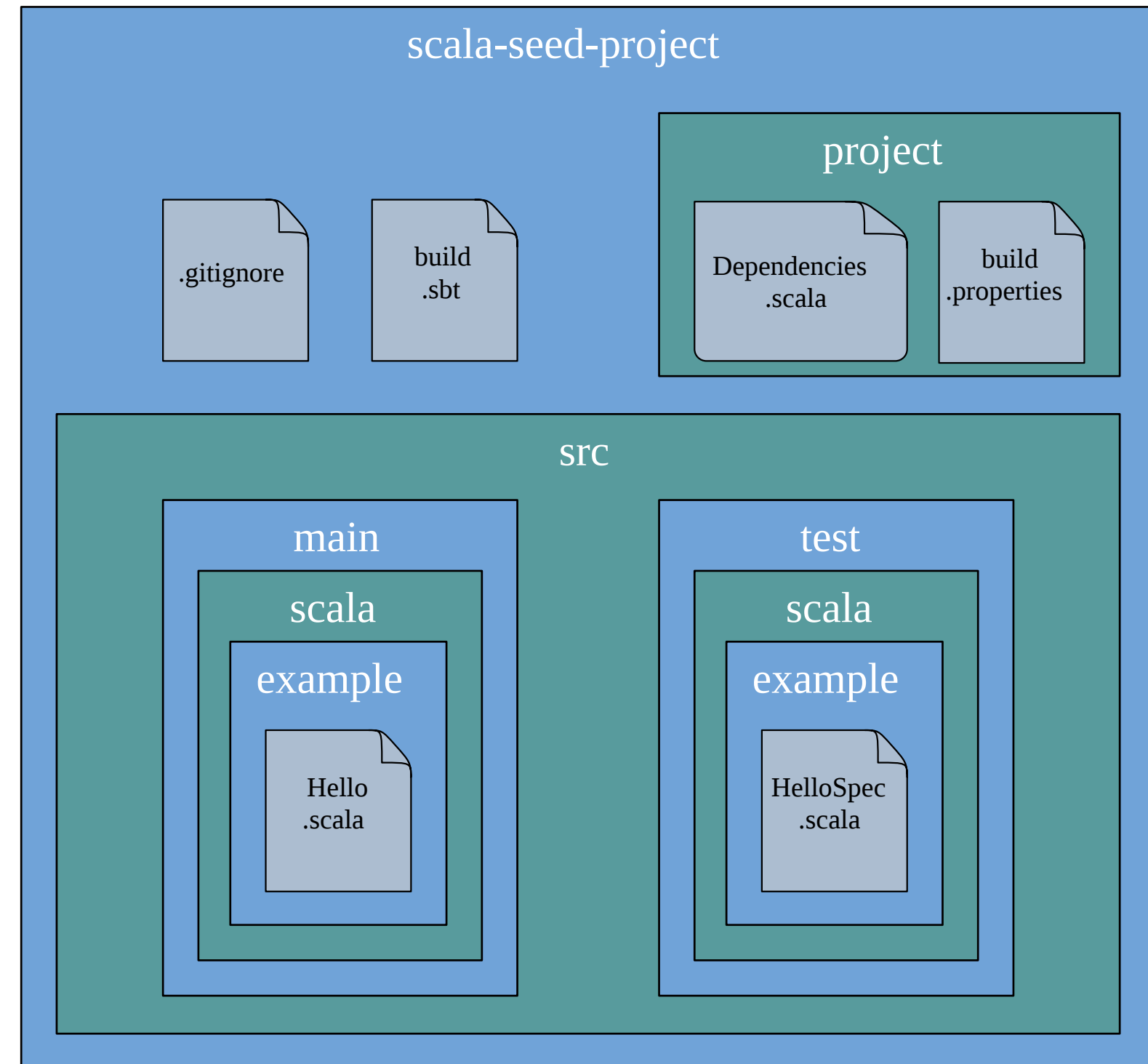
```
trimAll(john)  
// res: Json = JsonObject(Map(  
//   "name"      -> JsonString("John Doe"),  
//   "age"       -> JsonNumber(25),  
//   "email"     -> JsonString("john@doe.com"),  
//   "address"  -> JsonObject(Map(  
//     "street-number" -> JsonNumber(12),  
//     "post-code"    -> JsonString("E16 4SR")  
//   ))
```

The background of the slide is a light blue gradient with a complex network of thin, light blue lines connecting various circular nodes. The nodes are also light blue, with some appearing slightly larger or more prominent than others. The network is distributed across the entire slide, creating a sense of connectivity and structure.

# JsonExercises.scala

# Recursive data structures

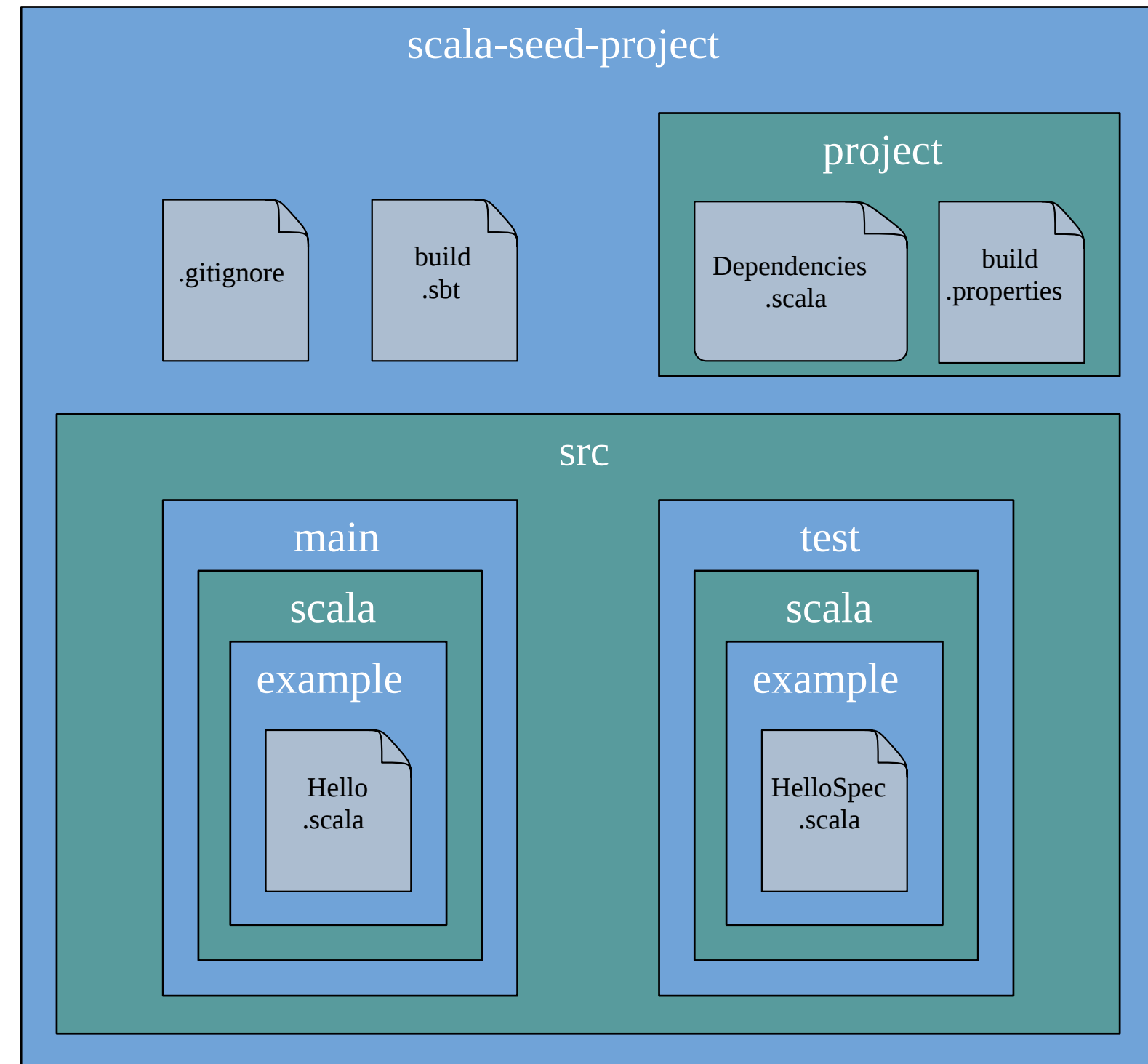
```
$ sbt new scala/scala-seed.g8
```



# File system: disk usage

```
$ sbt new scala/scala-seed.g8
```

```
$ cd scala-seed-project
$ du -b .
249  ./project
344  ./src/test/scala/example
440  ./src/test/scala
536  ./src/test
234  ./src/main/scala/example
330  ./src/main/scala
426  ./src/main
1090 ./src
1986 .
```



# Imperative approach

```
import java.io.File

def diskUsage(file: File): Long =
  ???
```

# Imperative approach

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory)
    ???
  else // normal file
    file.length()
```

# Imperative approach

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    var total = 0L

    for (child <- file.listFiles())
      total += child.length()

    total
  } else
    file.length()
```

# Imperative approach

```
import java.io.File

def diskUsage(file: File): Long = {
  var total = file.length()

  if(file.isDirectory) {
    for (child <- file.listFiles())
      total += child.length()
  }

  total
}
```



# Imperative approach

```
import java.io.File
import scala.collection.mutable

def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```

# Imperative approach

```
import java.io.File
import scala.collection.mutable

def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)


  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```

total



# Imperative approach

```
import java.io.File
import scala.collection.mutable

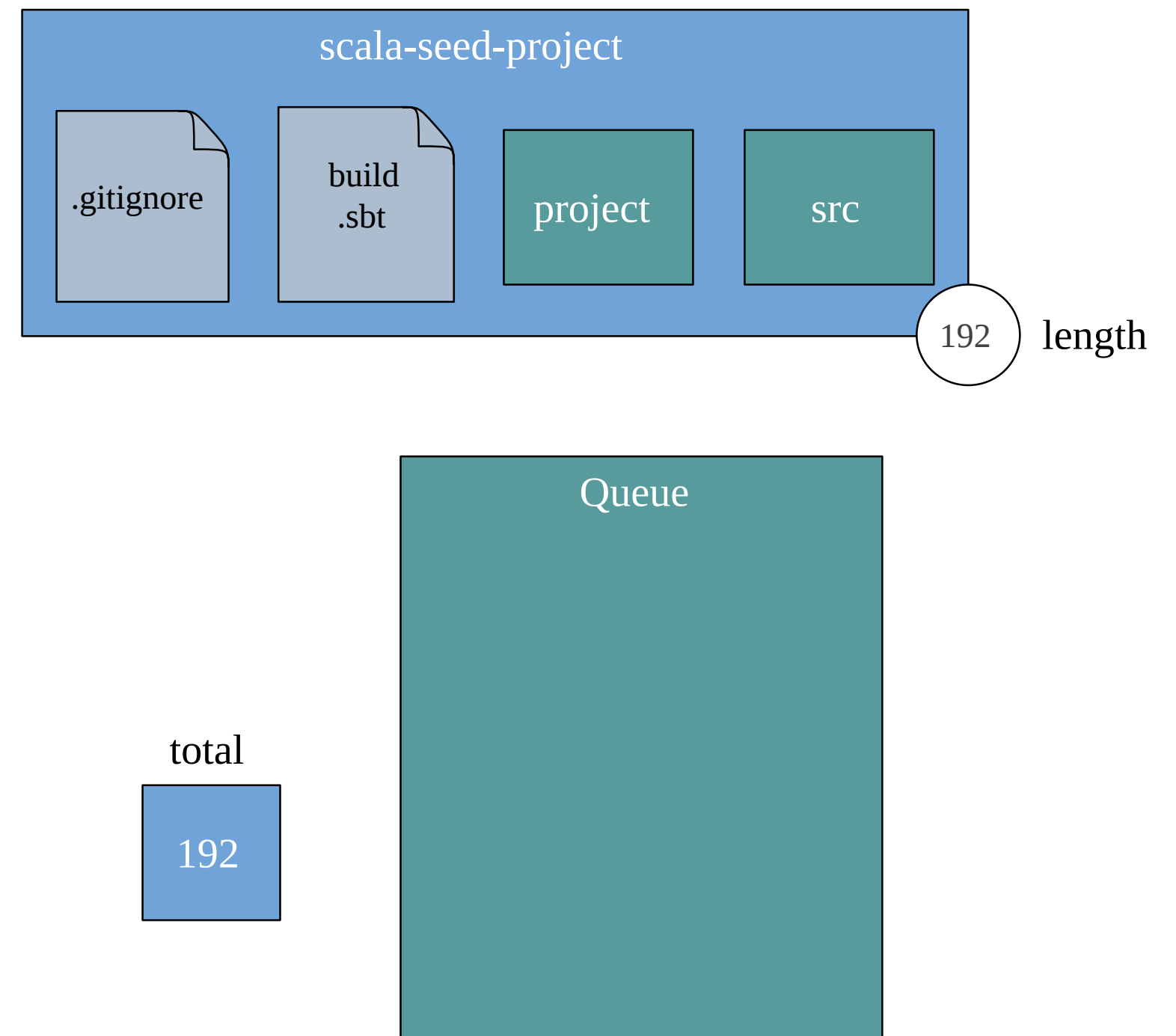
def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```



# Imperative approach

```
import java.io.File
import scala.collection.mutable

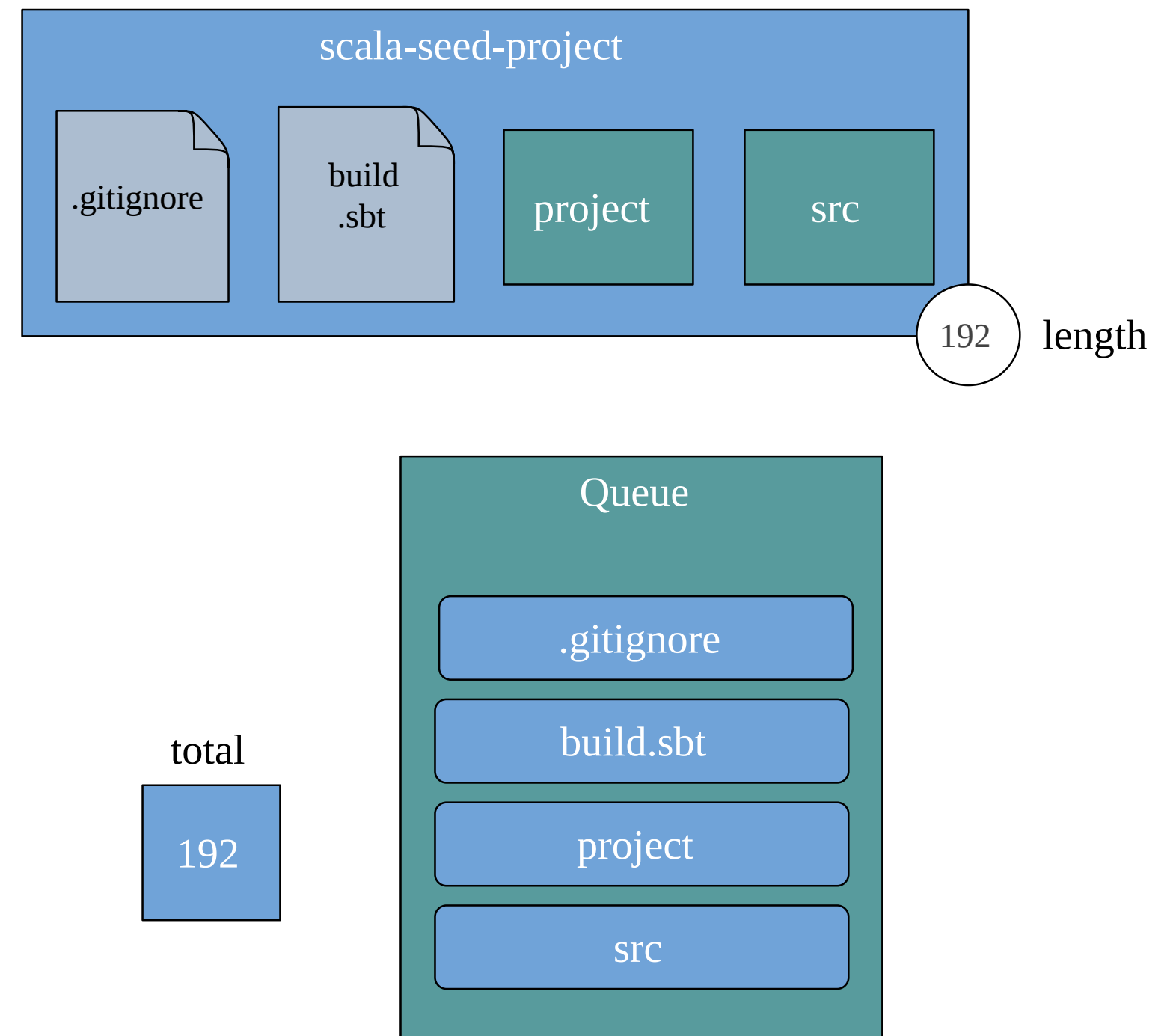
def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```



# Imperative approach

```
import java.io.File
import scala.collection.mutable

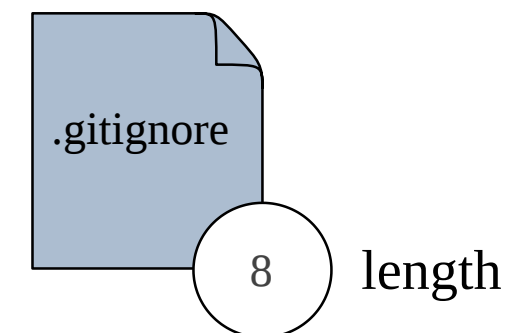
def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

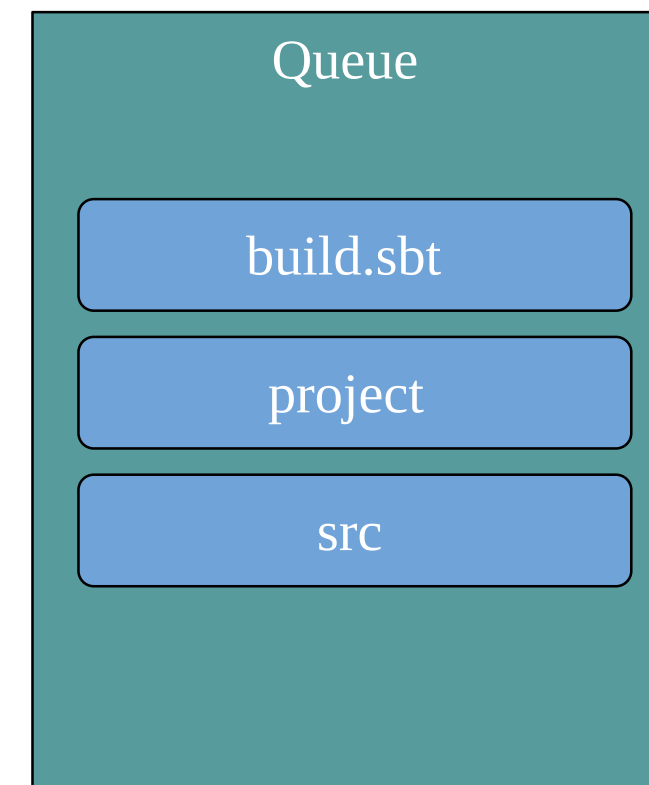
    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```



total

200



# Imperative approach

```
import java.io.File
import scala.collection.mutable

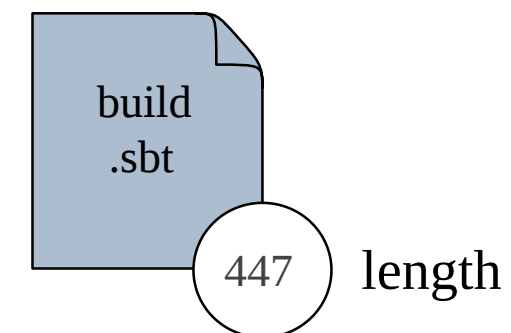
def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

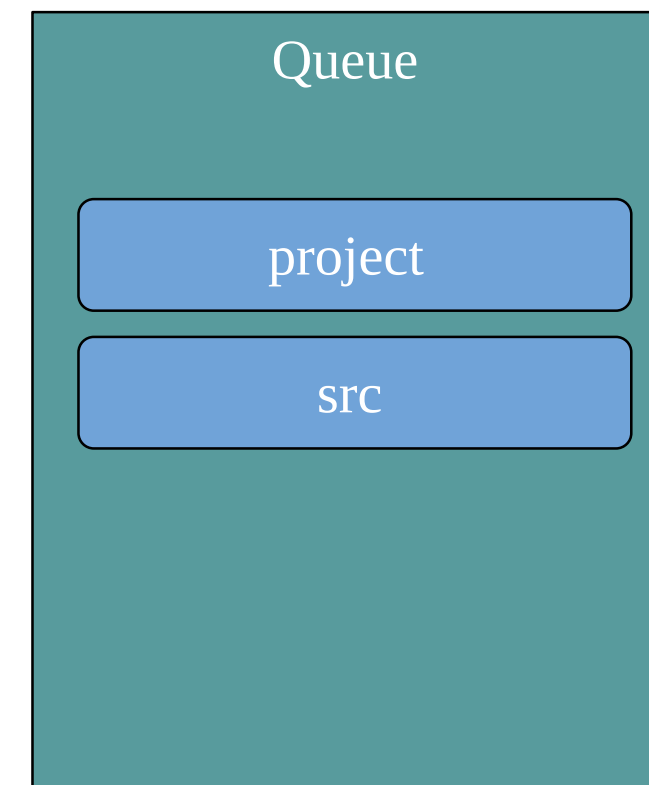
    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```



total  
647



# Imperative approach

```
import java.io.File
import scala.collection.mutable

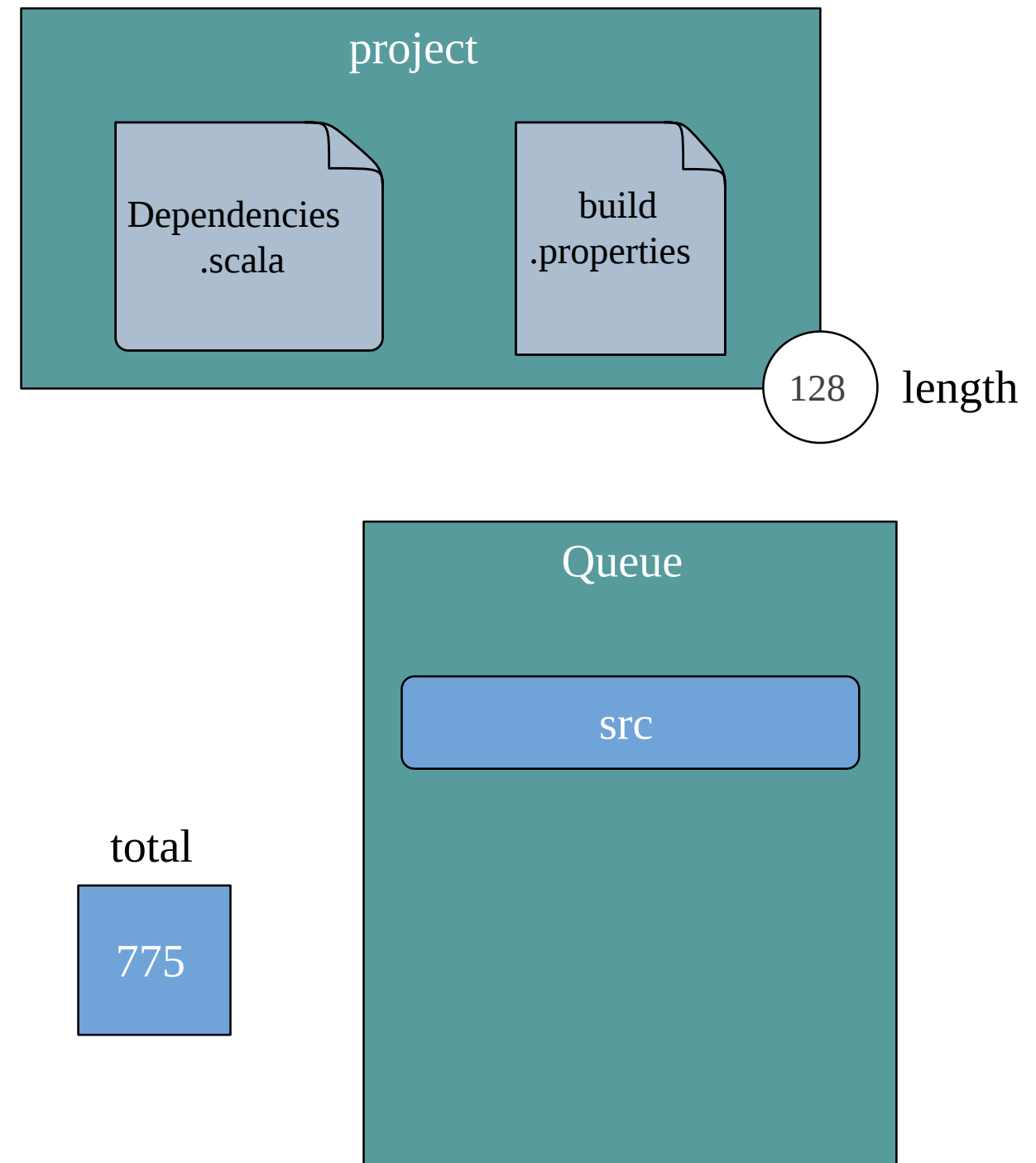
def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```



# Imperative approach

```
import java.io.File
import scala.collection.mutable

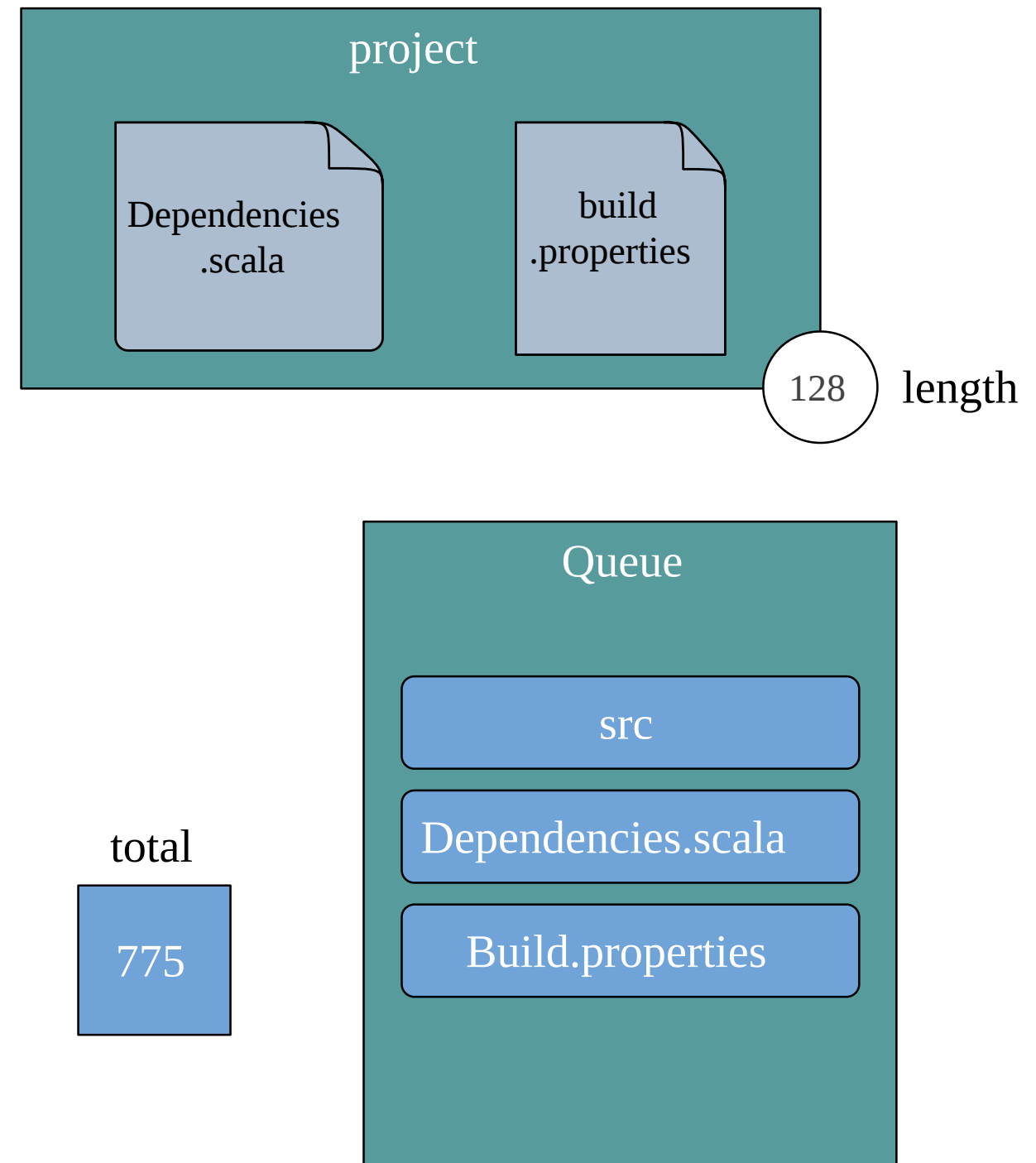
def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```





# Recursive approach

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory)
    ???
  else
    ???
```

# Recursive approach

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage: Long = ???

    file.length() + childrenDiskUsage
  } else
    file.length()
```

# Recursive approach

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles      // Array[File]
      .map(diskUsage) // Array[Long]
      .sum           // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```

# Imperative

```
import java.io.File
import scala.collection.mutable

def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```

# Recursive

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles
      .map(diskUsage)
      .sum

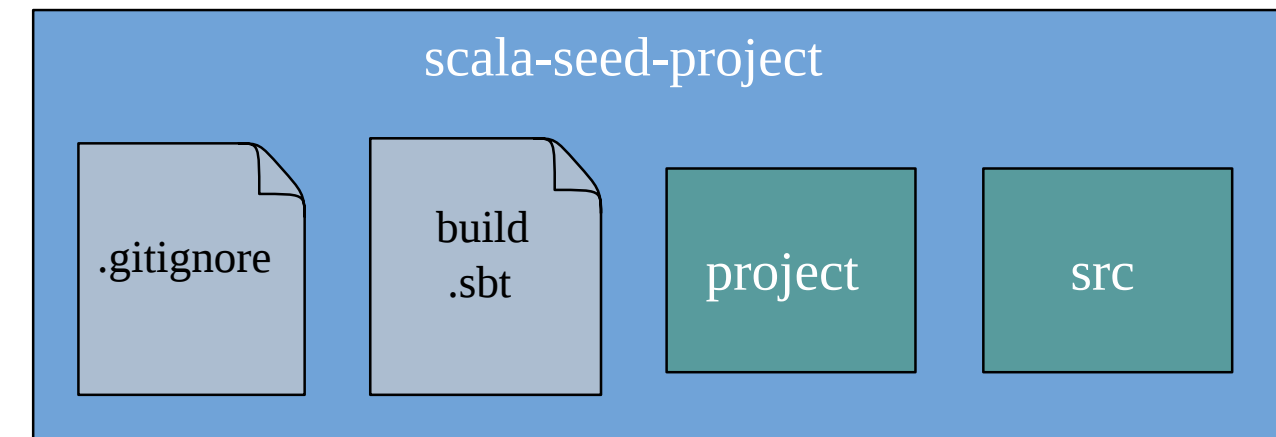
    file.length() + childrenDiskUsage
  } else
    file.length()
```

# File system: disk usage

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles           // Array[File]
      .map(diskUsage)     // Array[Long]
      .sum                 // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```

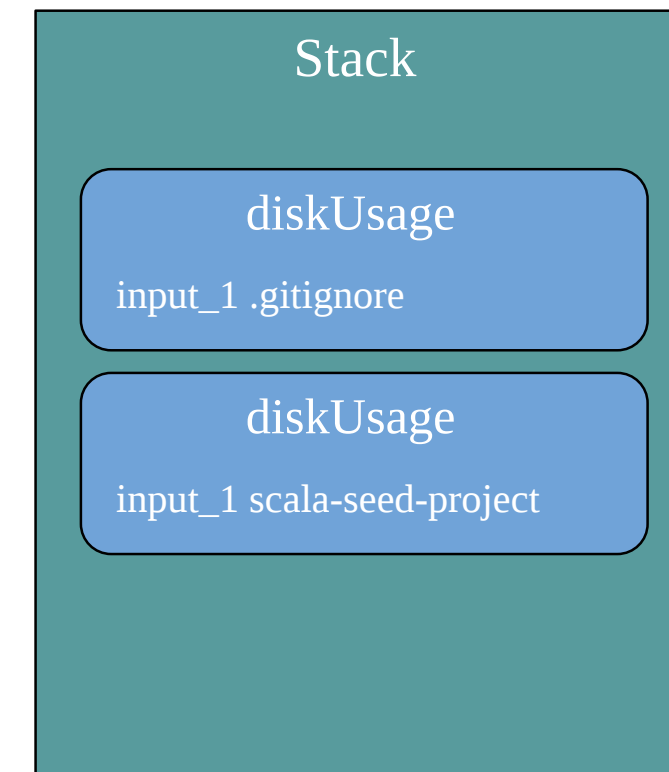


# File system: disk usage

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles           // Array[File]
      .map(diskUsage)     // Array[Long]
      .sum                // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```

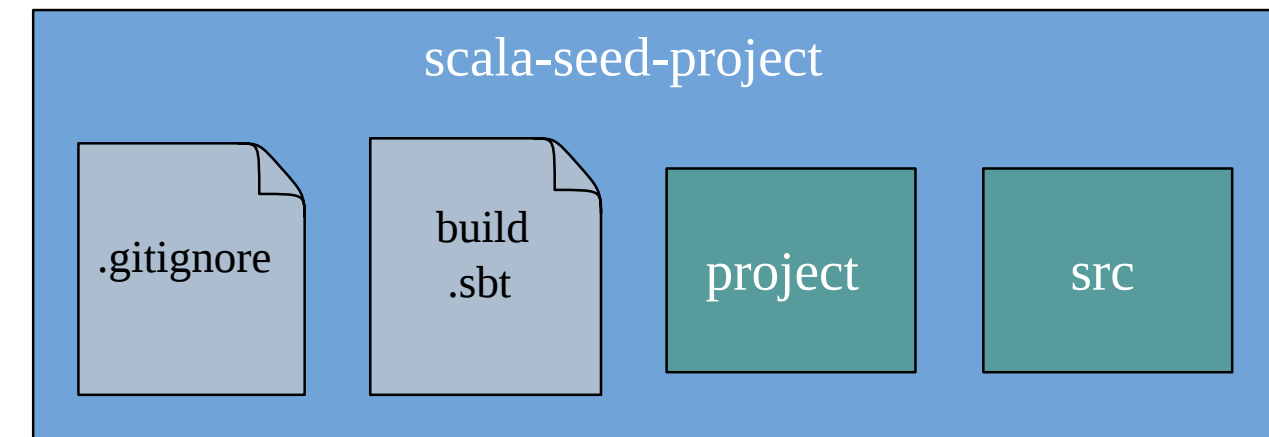


# File system: disk usage

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles           // Array[File]
      .map(diskUsage)     // Array[Long]
      .sum                 // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```

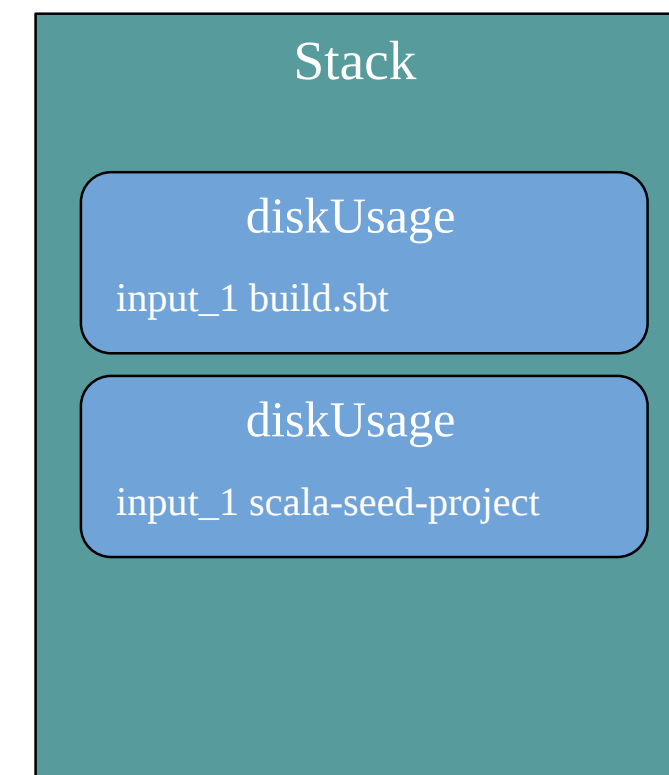
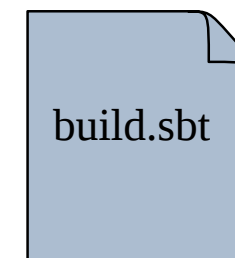


# File system: disk usage

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles          // Array[File]
      .map(diskUsage)     // Array[Long]
      .sum                // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```



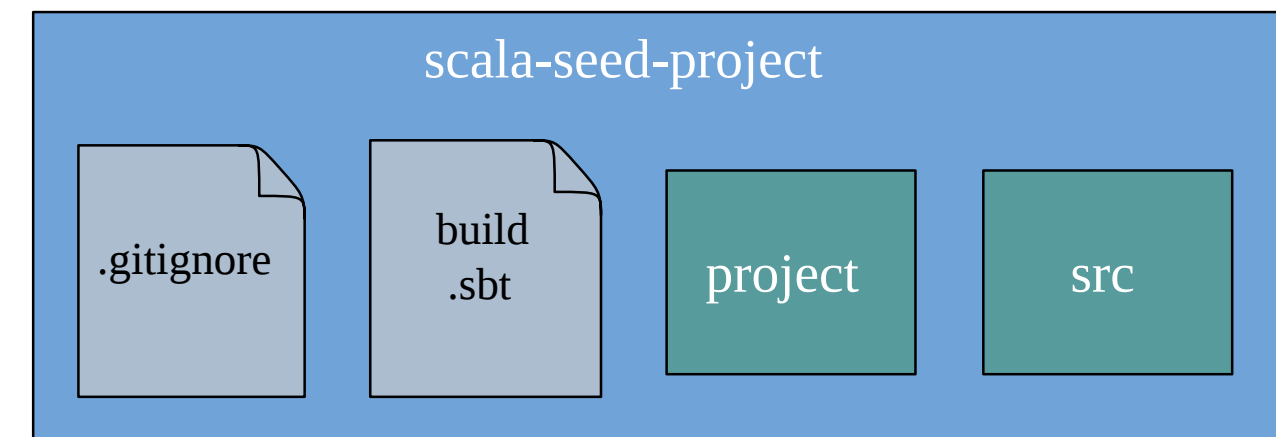


# File system: disk usage

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles           // Array[File]
      .map(diskUsage)     // Array[Long]
      .sum                 // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```

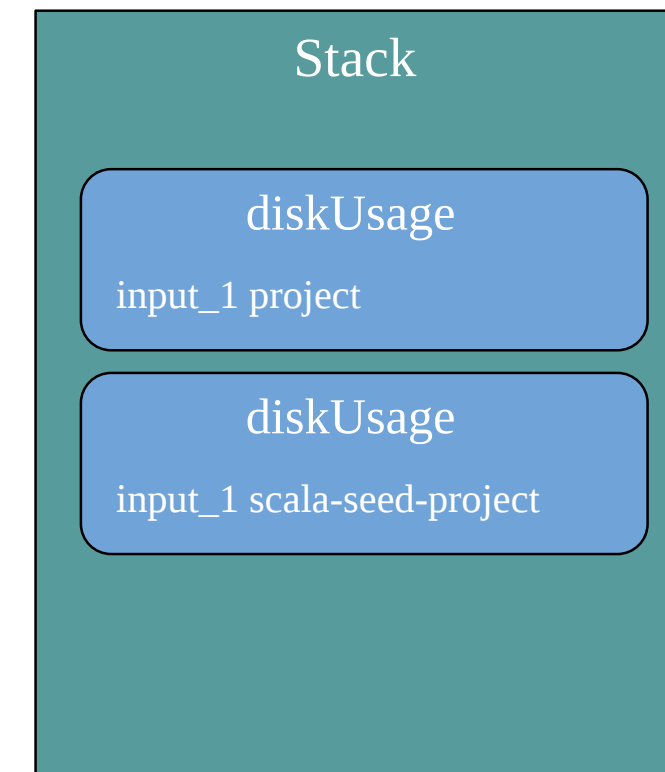
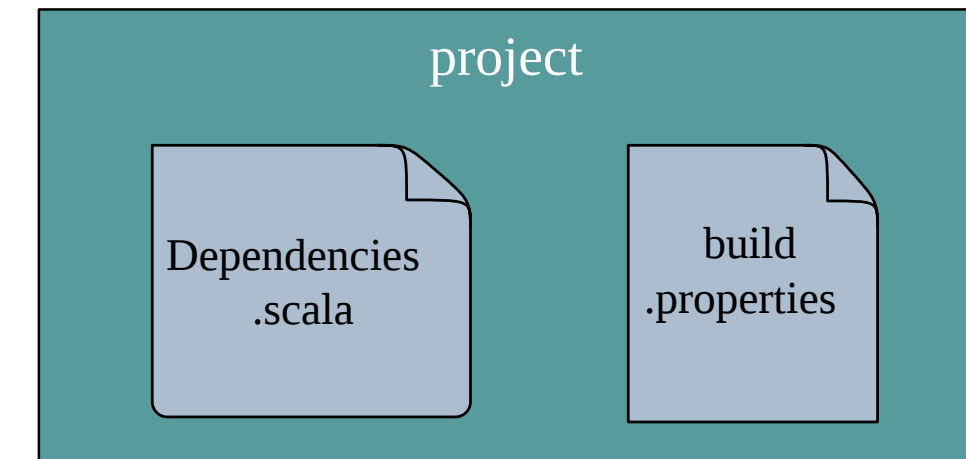


# File system: disk usage

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles          // Array[File]
      .map(diskUsage)     // Array[Long]
      .sum                // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```

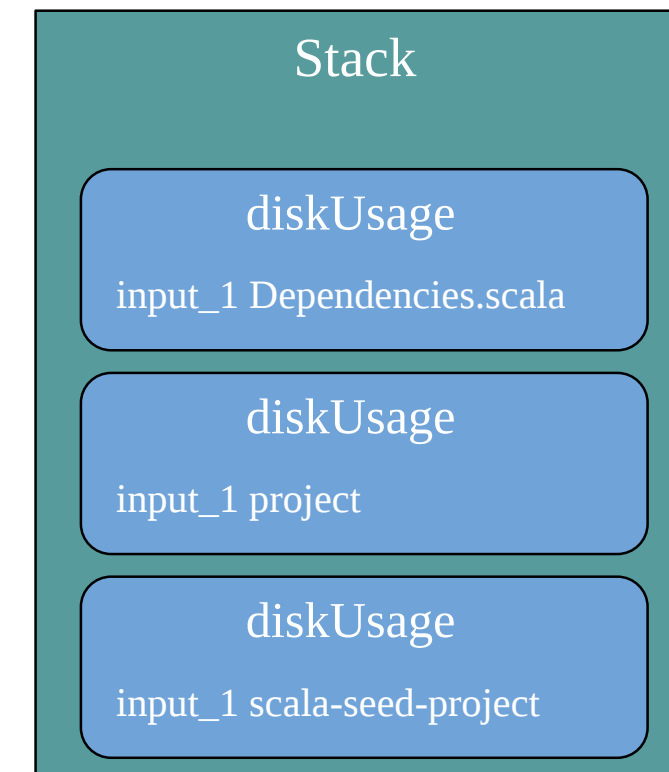
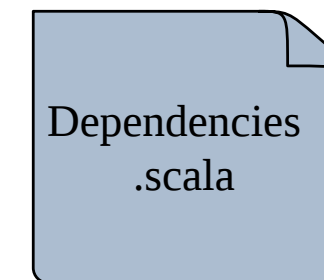


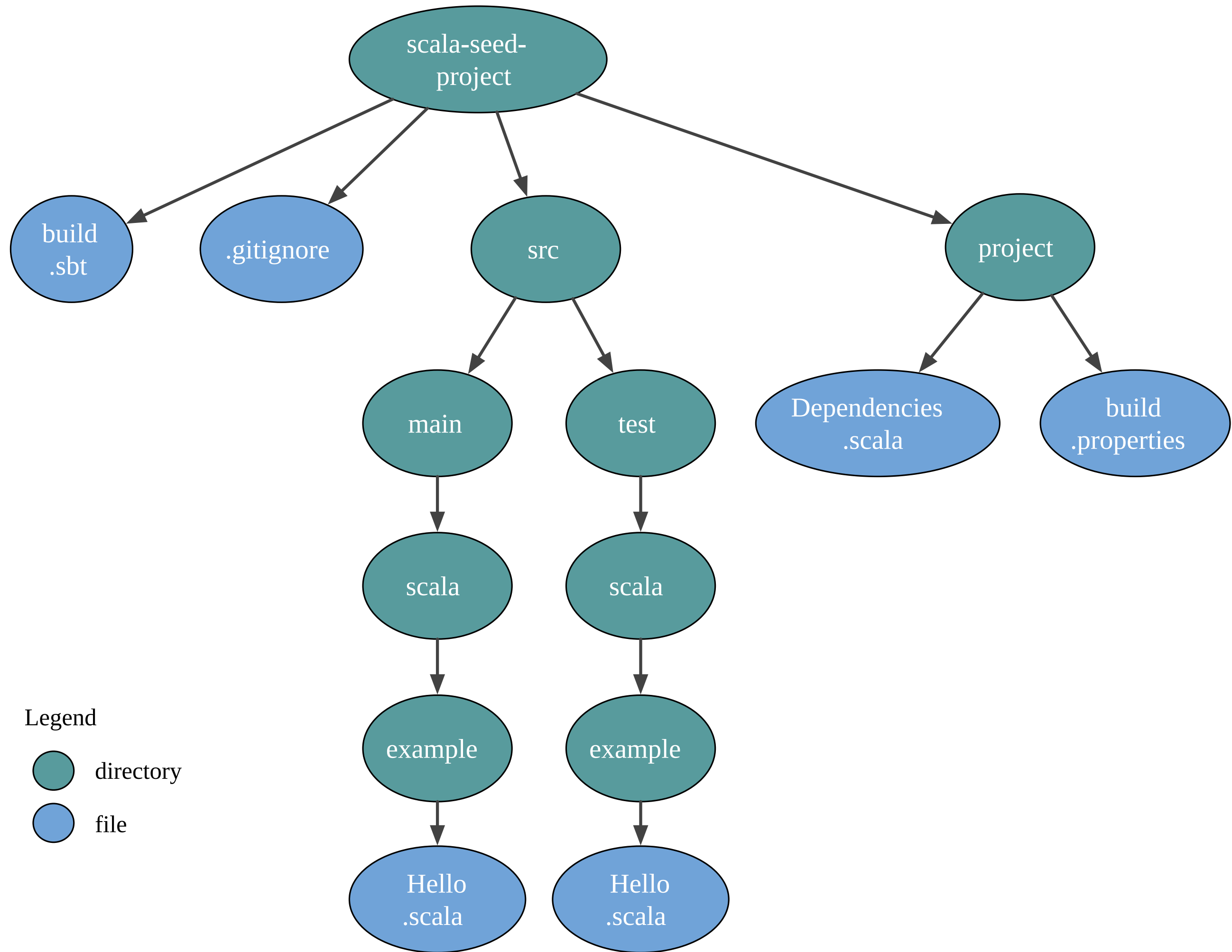
# File system: disk usage

```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles           // Array[File]
      .map(diskUsage)     // Array[Long]
      .sum                 // Long

    file.length() + childrenDiskUsage
  } else
    file.length()
```

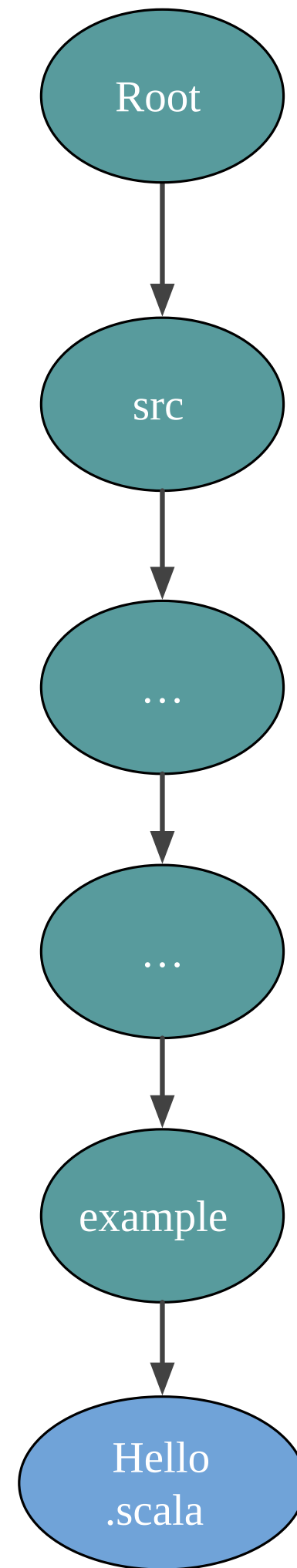




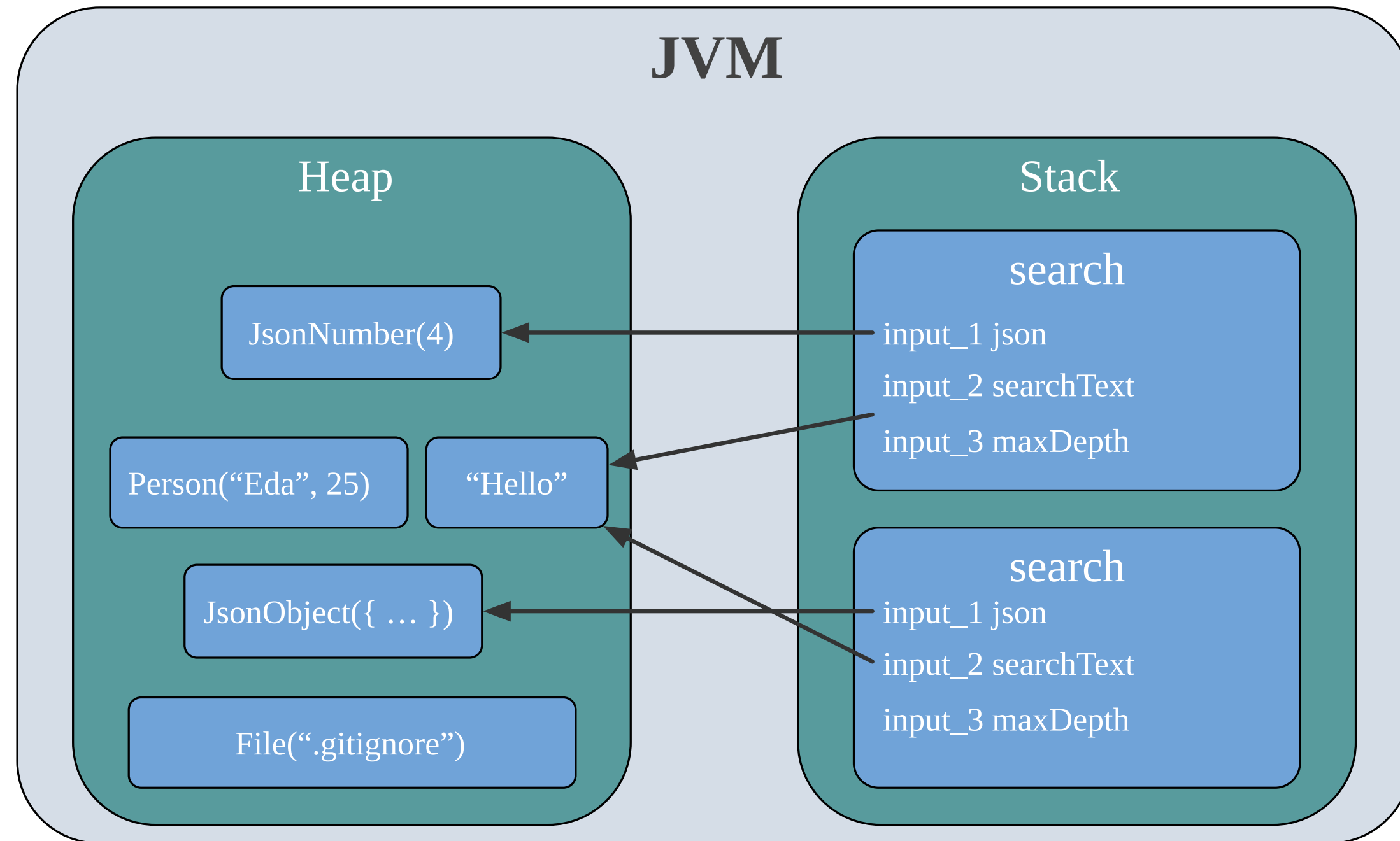
Legend

● directory

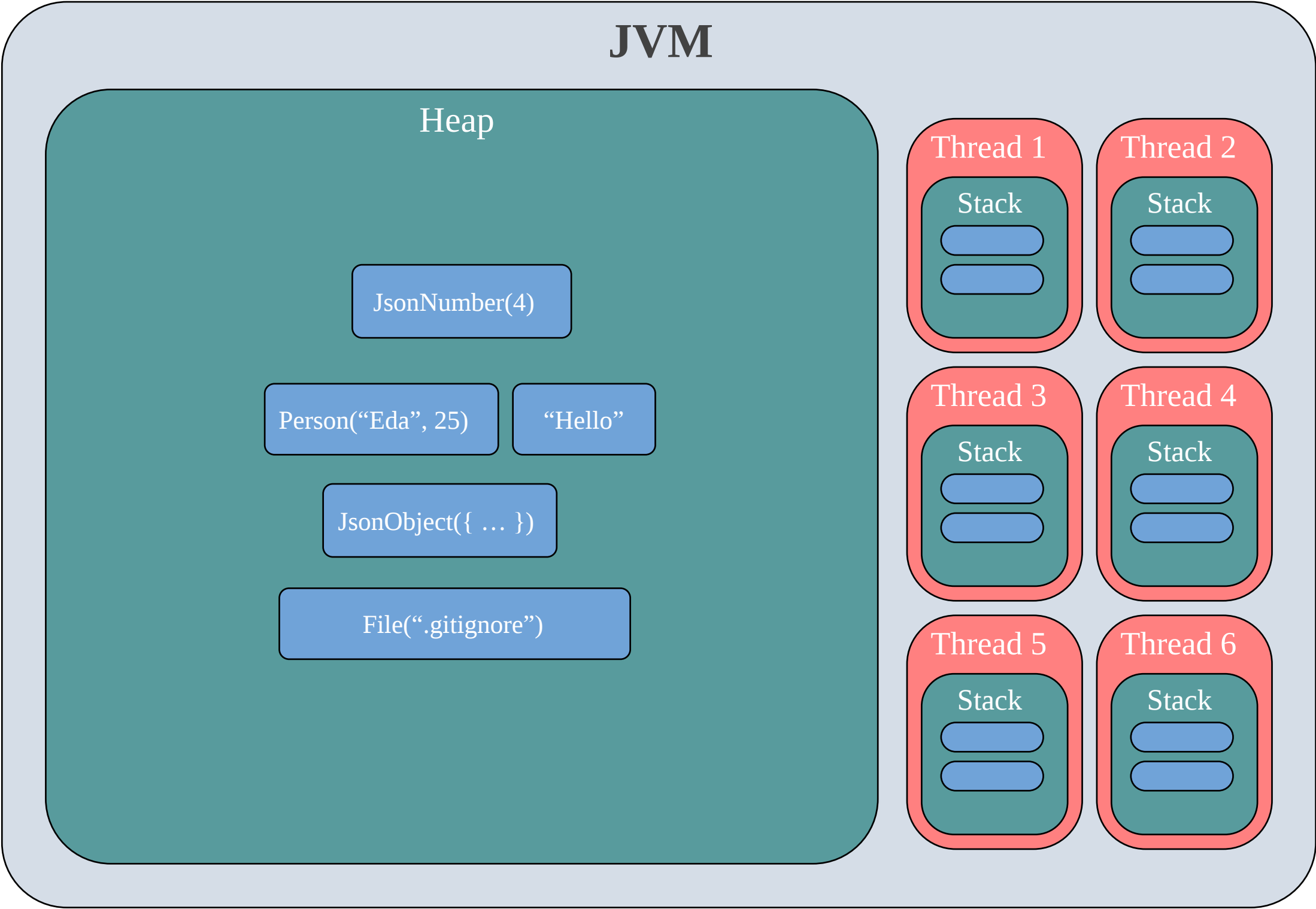
● file



# JVM memory model



# JVM memory model



# Explicit data structure

```
import java.io.File
import scala.collection.mutable

def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```

# Implicit data structure

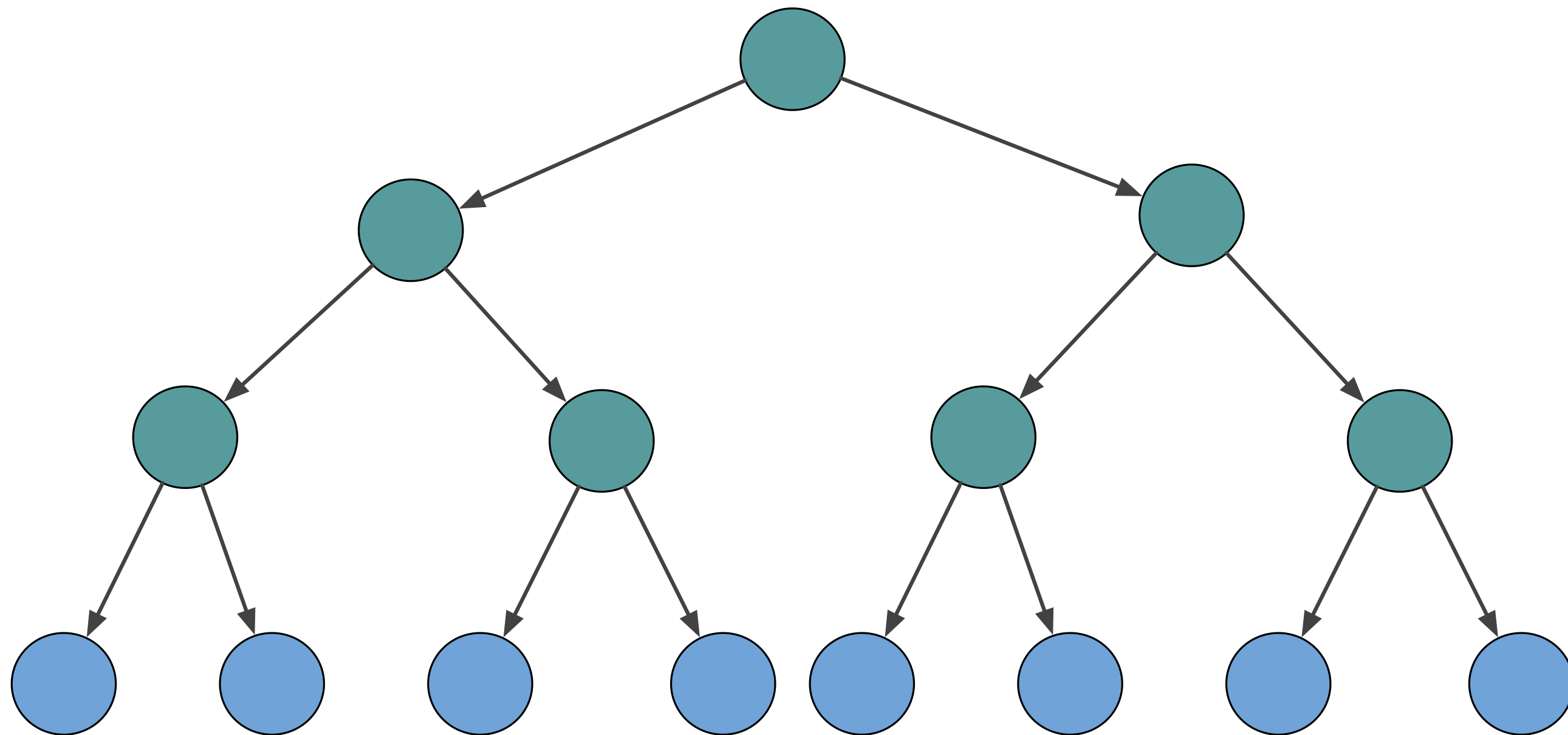
```
import java.io.File

def diskUsage(file: File): Long =
  if(file.isDirectory) {
    val childrenDiskUsage = file
      .listFiles           // Array[File]
      .map(diskUsage)      // Array[Long]
      .sum                 // Long

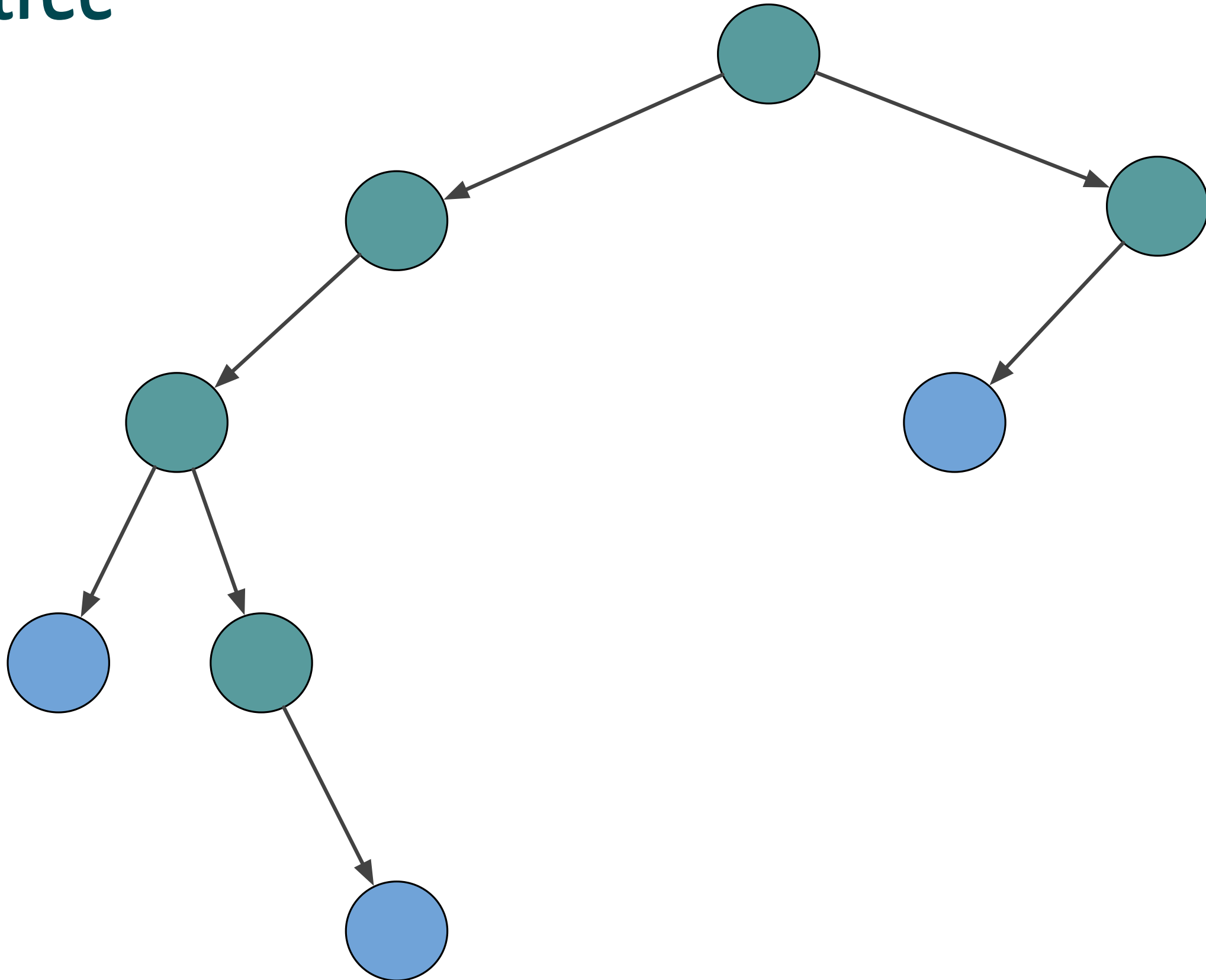
    file.length() + childrenDiskUsage
  } else
    file.length()
```



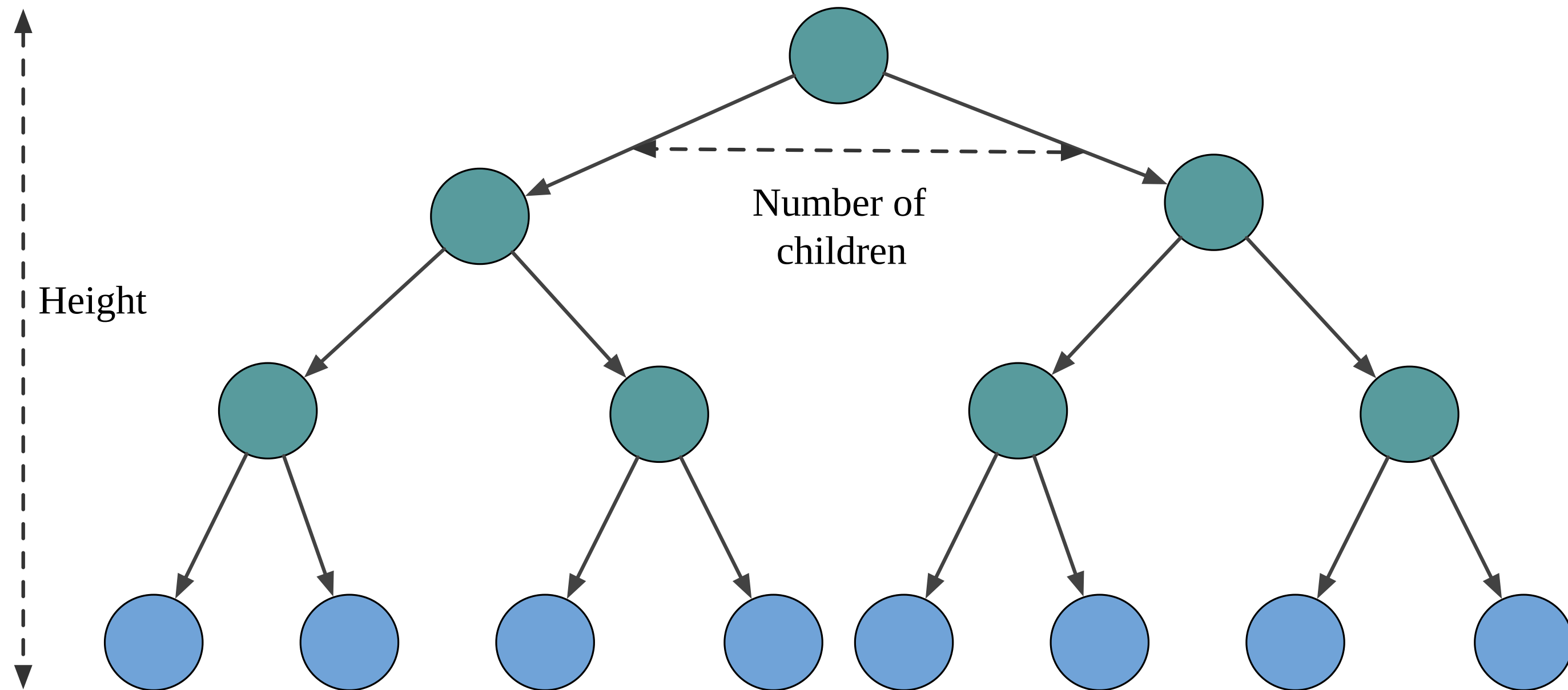
# How many elements?



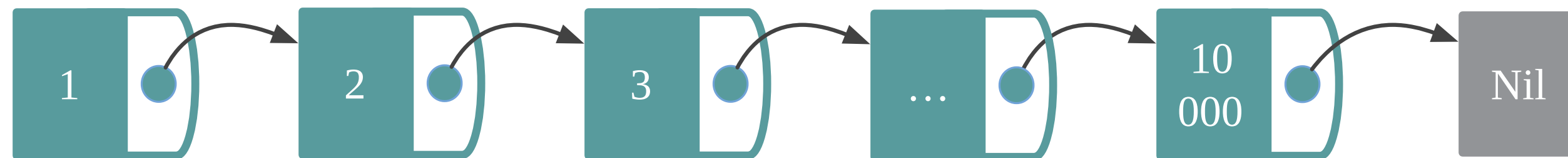
# Unbalanced tree



# How many elements?



# List



# Recursive sum

```
def sum(numbers: List[Int]): Int =  
  ???
```

# Recursive sum

```
def sum(numbers: List[Int]): Int =  
  numbers match {  
    case Nil          => ???  
    case head :: tail => ???  
  }
```

```
enum List[+A] {  
  case Nil  
  case ::(head: A, tail: List[A])  
}
```

# Recursive sum

```
def sum(numbers: List[Int]): Int =  
  numbers match {  
    case Nil          => ???  
    case head :: tail => ???  
  }
```

```
def sum(numbers: List[Int]): Int =  
  numbers match {  
    case Nil          => ???  
    case ::(head, tail) => ???  
  }
```

```
enum List[+A] {  
  case Nil  
  case ::(head: A, tail: List[A])  
}
```

# Recursive sum

```
def sum(numbers: List[Int]): Int =  
  numbers match {  
    case Nil => 0  
    case head :: tail => sum(tail) + head  
  }
```



# Recursive sum

```
def sum(numbers: List[Int]): Int =  
  numbers match {  
    case Nil => 0  
    case head :: tail => sum(tail) + head  
  }
```

```
sum(List(1,2,3,4,5))  
// res47: Int = 15
```

```
val largeList = List.range(1, 10000)  
// largeList: List[Int] = List(1,2,3,4,...,10000)  
  
sum(largeList)  
// java.lang.StackOverflowError
```

# Recursive contains

```
def contains(list: List[Int], number: Int): Boolean =  
  list match {  
    case Nil => false  
    case head :: tail =>  
      if (head == number)  
        true  
      else  
        contains(tail, number)  
  }
```

```
contains(List(1,2,3,4), 3)  
// res49: Boolean = true
```

```
contains(List(1,2,3,4), 5)  
// res50: Boolean = false
```

# Recursive contains

```
def contains(list: List[Int], number: Int): Boolean =  
  list match {  
    case Nil => false  
    case head :: tail =>  
      if (head == number)  
        true  
      else  
        contains(tail, number)  
  }
```

```
contains(List(1,2,3,4), 3)  
// res52: Boolean = true
```

```
contains(List(1,2,3,4), 5)  
// res53: Boolean = false
```

# Recursive contains

```
def contains(list: List[Int], number: Int): Boolean =  
  list match {  
    case Nil => false  
    case head :: tail =>  
      if (head == number)  
        true  
      else  
        contains(tail, number)  
  }
```

```
val largeList = List.range(1, 10000)  
// largeList: List[Int] = List(1,2,3,4,...,10000)  
  
contains(largeList, 5)  
// res: Boolean = true
```

# Recursive contains

```
def contains(list: List[Int], number: Int): Boolean =  
  list match {  
    case Nil => false  
    case head :: tail =>  
      if (head == number)  
        true  
      else  
        contains(tail, number)  
  }
```

```
val largeList = List.range(1, 10000)  
// largeList: List[Int] = List(1,2,3,4,...,10000)  
  
contains(largeList, 5)  
// res: Boolean = true  
  
contains(largeList, -1)  
// res: Boolean = false
```

# Recursive functions

```
def contains(list: List[Int], number: Int): Boolean =  
  list match {  
    case Nil => false  
    case head :: tail =>  
      if (head == number)  
        true  
      else  
        contains(tail, number)  
  }
```

```
def sum(list: List[Int]): Int =  
  list match {  
    case Nil => 0  
    case head :: tail =>  
      sum(tail) + head  
  }
```

# Recursive functions

```
def contains(list: List[Int], number: Int): Boolean =  
  list match {  
    case Nil => false  
    case head :: tail =>  
      if (head == number)  
        true  
      else  
        contains(tail, number)  
  }
```

```
def sum(list: List[Int]): Int =  
  list match {  
    case Nil => 0  
    case head :: tail =>  
      head + sum(tail)  
  }
```

# Recursive functions

```
def contains(list: List[Int], number: Int): Boolean =  
  list match {  
    case Nil => false  
    case head :: tail =>  
      if (head == number)  
        true  
      else  
        contains(tail, number)  
  }
```

```
def sum(list: List[Int]): Int =  
  list match {  
    case Nil => 0  
    case head :: tail =>  
      val rest = sum(tail)  
      head + rest  
  }
```



# Tail recursion

```
import scala.annotation.tailrec

@tailrec
def contains(list: List[Int], number: Int): Boolean
=
  list match {
    case Nil => false
    case head :: tail =>
      if (head == number)
        true
      else
        contains(tail, number)
  }
```

```
import scala.annotation.tailrec

@tailrec
def sum(list: List[Int]): Int =
  list match {
    case Nil => 0
    case head :: tail =>
      sum(tail) + head
  }

// [error] could not optimize @tailrec
// annotated method sum:
// it contains a recursive call not in
// tail position
```

# Tail recursive sum

```
import scala.annotation.tailrec

@tailrec
def sum(numbers: List[Int], state: Int): Int =
  numbers match {
    case Nil          => state
    case head :: tail => sum(tail, state + head)
  }
```

```
sum(List(1,2,3,4,5), 0)
// res: Int = 15

sum(List.range(1, 10000), 0)
// res: Int = 49995000
```

# Tailrec vs imperative

```
import scala.annotation.tailrec

@tailrec
def sum(numbers: List[Int], state: Int): Int =
  numbers match {
    case Nil          => state
    case head :: tail => sum(tail, state + head)
  }
```

```
def sum(numbers: List[Int]): Int = {
  var state = 0

  for (number <- numbers)
    state += number

  state
}
```

# Tailrec vs imperative

```
import scala.annotation.tailrec

@tailrec
def sum(numbers: List[Int], state: Int): Int =
  numbers match {
    case Nil          => state
    case head :: tail => sum(tail, state + head)
  }

def sum(numbers: List[Int]): Int =
  sum(numbers, state = 0)
```

```
def sum(numbers: List[Int]): Int = {
  var state = 0

  for (number <- numbers)
    state += number

  state
}
```

# Hide tailrec implementation

## Private method

```
@tailrec
private def sum(numbers: List[Int], state: Int): Int =
  numbers match {
    case Nil => state
    case head :: tail => sum(tail, state + head)
  }

def sum(numbers: List[Int]): Int =
  sum(numbers, state = 0)
```

## Nested method

```
def sum(numbers: List[Int]): Int = {
  @tailrec
  def go(numbers: List[Int], state: Int): Int =
    numbers match {
      case Nil => state
      case head :: tail => sum(tail, state + head)
    }

  go(numbers, state = 0)
}
```



StackSafeRecursiveExercises.scala

# Summary

- Recursive functions are convenient to traverse recursive data structures
- Stack-safety concerns
- Solutions:
  - limit the depth
  - tailrec annotation

# Tail recursive sum

```
def sum(numbers: List[Int]): Int = {  
  @tailrec  
  def go(numbers: List[Int], state: Int): Int =  
    numbers match {  
      case Nil => state  
      case head :: tail => sum(tail, state + head)  
    }  
  go(numbers, state = 0)  
}
```



# Imperative diskUsage

```
import java.io.File
import scala.collection.mutable

def diskUsage(input: File): Long = {
  var total = 0L
  val queue = mutable.Queue(input)

  while (queue.nonEmpty) {
    val file = queue.dequeue()

    total += file.length()

    if(file.isDirectory)
      queue.addAll(file.listFiles())
  }

  total
}
```