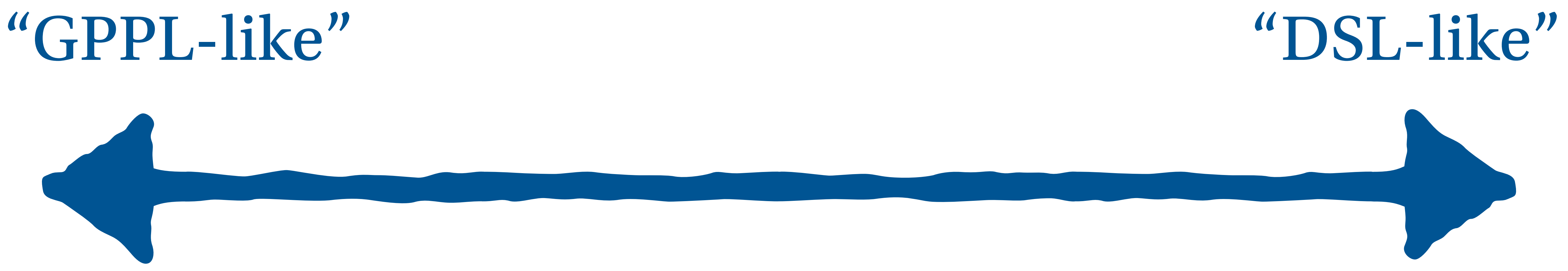
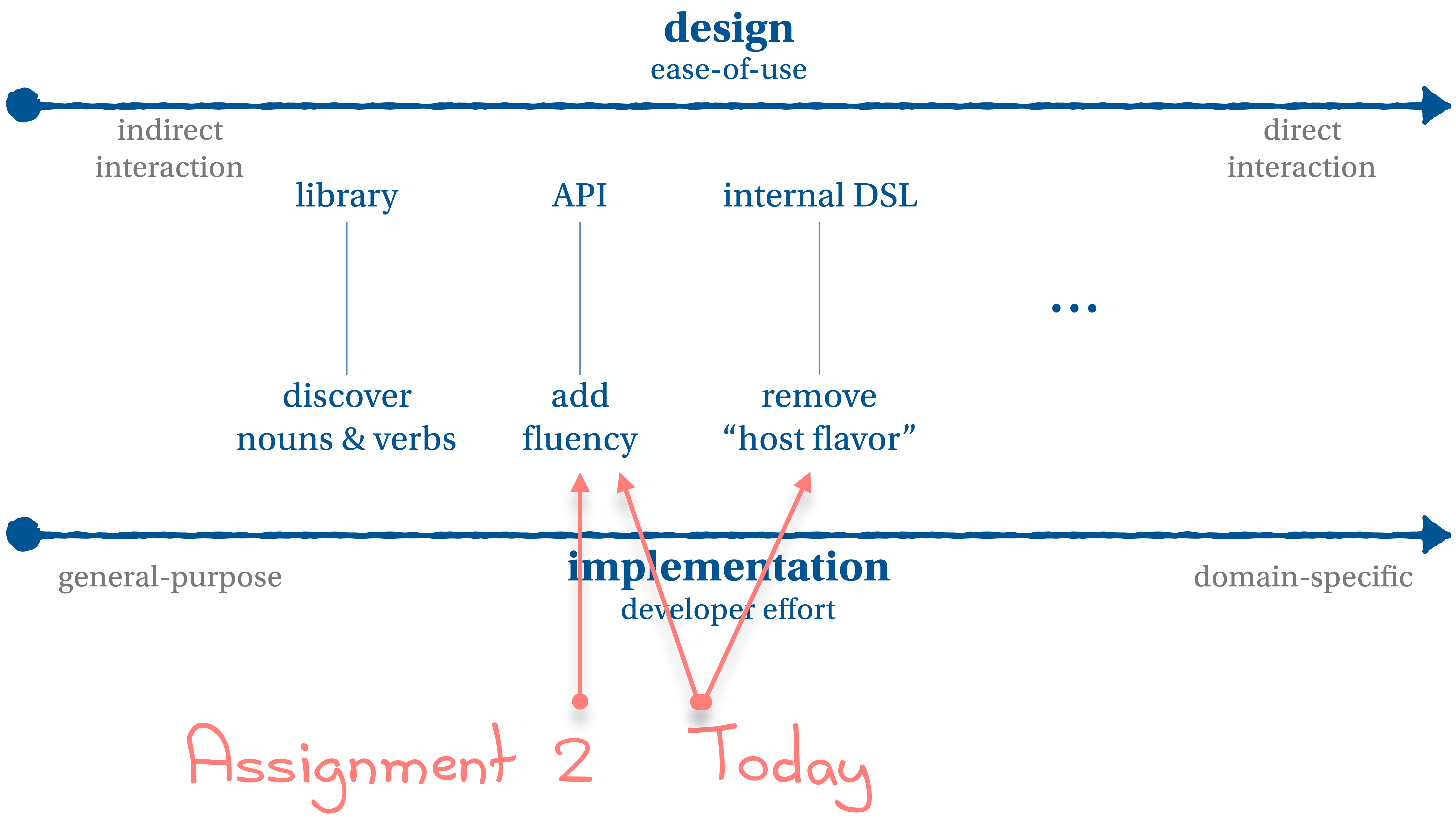
Recall: It's a spectrum, not a binary



How precise are our answers to these questions?

1. Is it a programming language?
2. What is the focus? What does the *domain expert* describe?
3. What is easy, difficult, impossible in this language?
   (relative to a general-purpose programming language)

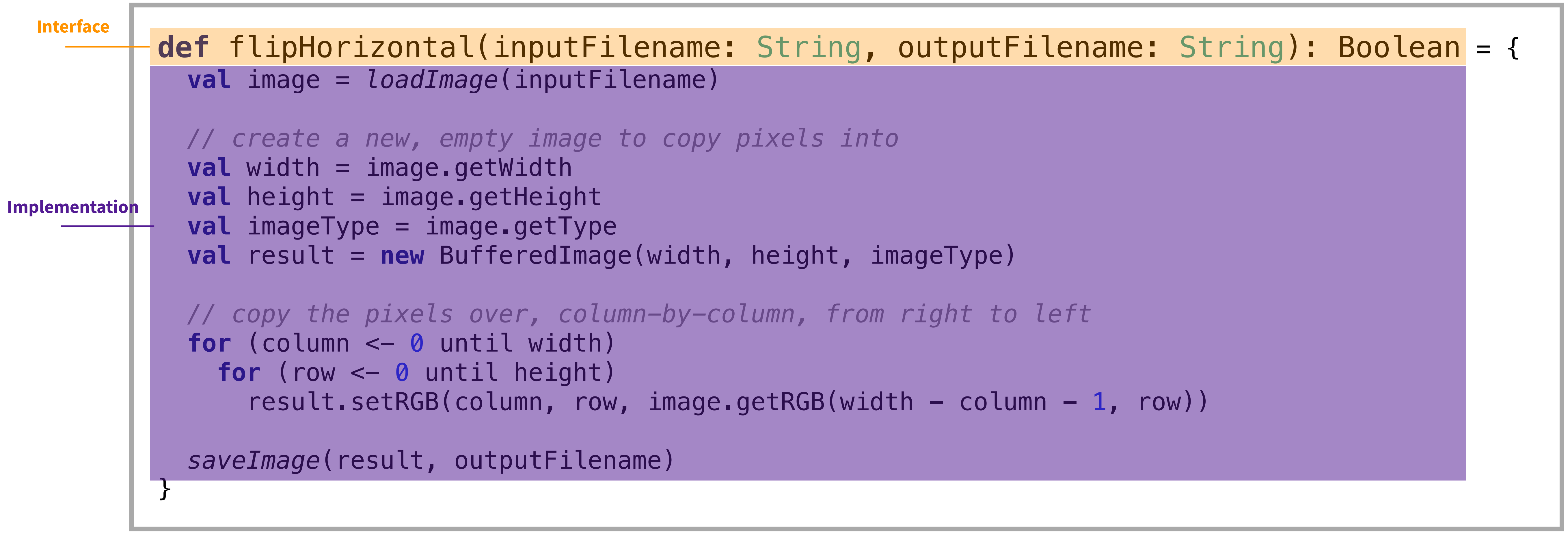
Today: Implementation techniques



First: some terminology

- **Interface**: *what* a piece of code can do
  An interface tells the user what the code can do, and doesn't require the user to know *how* the code does it.

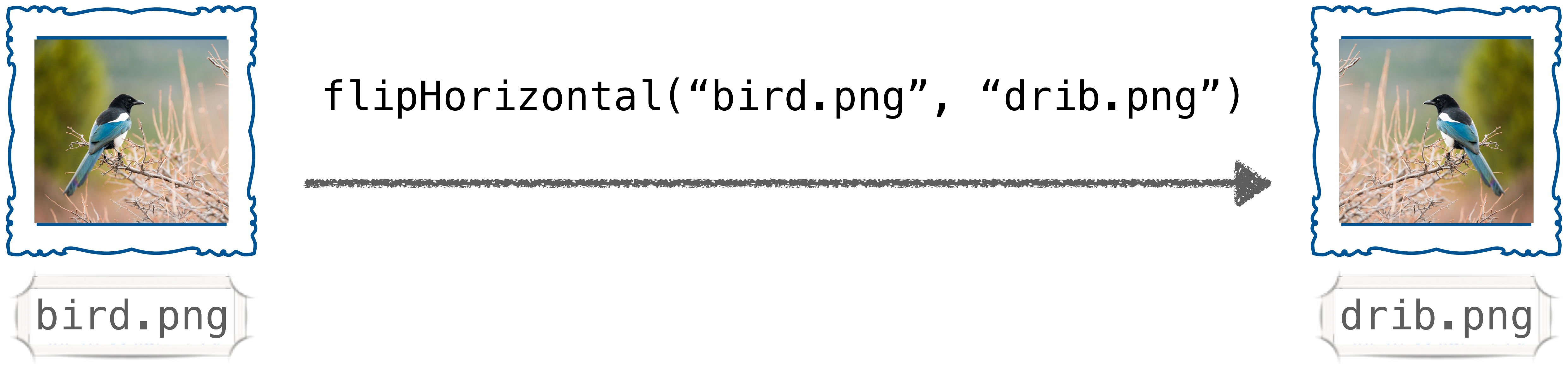
- **Implementation**: *how* a piece of code works
  Users shouldn't need to know or rely on this information, to use the code.



```
def flipHorizontal(inputFilename: String, outputFilename: String): Boolean = {
  val image = loadImage(inputFilename)

  // create a new, empty image to copy pixels into
  val width = image.getWidth
  val height = image.getHeight
  val imageType = image.getType
  val result = new BufferedImage(width, height, imageType)

  // copy the pixels over, column-by-column, from right to left
  for (column <- 0 until width)
    for (row <- 0 until height)
      result.setRGB(column, row, image.getRGB(width - column - 1, row))

  saveImage(result, outputFilename)
}
```

Our library, before the assignment

```
flipHorizontal(inputFilename, outputFilename)
flipVertical(inputFilename, outputFilename)
rotateLeft(inputFilename, outputFilename)
rotateRight(inputFilename, outputFilename)
grayScale(inputFilename, outputFilename)
```



```
flipHorizontal(“bird.png”, “drib.png”)
```

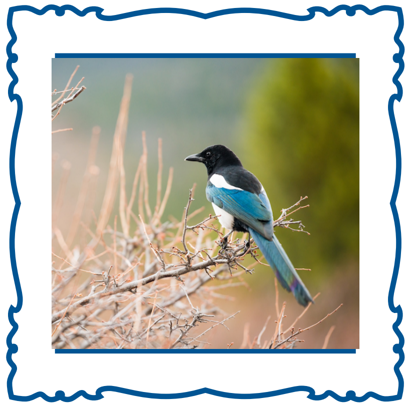
photo credit: Natasha Miller @ unsplash

Our library, after the assignment

```
loadImage(filename) => picture
flipHorizontal(picture) => picture
flipVertical(picture) => picture
rotateLeft(picture) => picture
rotateRight(picture) => picture
grayScale(picture) => picture
saveImage(picture, filename)
```

bird.png

```
saveImage(flipHorizontal(loadImage("bird.png")), "drib.png")
```



drib.png

This library is more fluent because it is **compositional**.

photo credit: Natasha Miller @ unsplash

Fluency: nesting *vs* chaining

They require different implementation techniques.

```
object PictureProgram extends App {
  val image = load(resource("/image.png"))
  val result =
    rotateLeft(
      grayScale(
        flipHorizontal(image)
      )
    )
  save(result, "output0.png")
}
```

Nested Calls
implemented with
composable functions

```
object PictureProgram extends App {
  val image = load(resource("/image.png"))

  image.flipHorizontal()
       .grayScale()
       .rotateLeft()
       .save("output0.png")
}
```

Chained calls
implemented with
composable methods

Program

=

Data + Operations

global variables & parameters

functions

Object

=

Data + **Operations**

field values method definitions

- An object is ready to use:
  - All its fields have values.
  - All its methods have been defined.
- An object can access its own fields & methods.
- Others can access an object through its interface.

Class

=

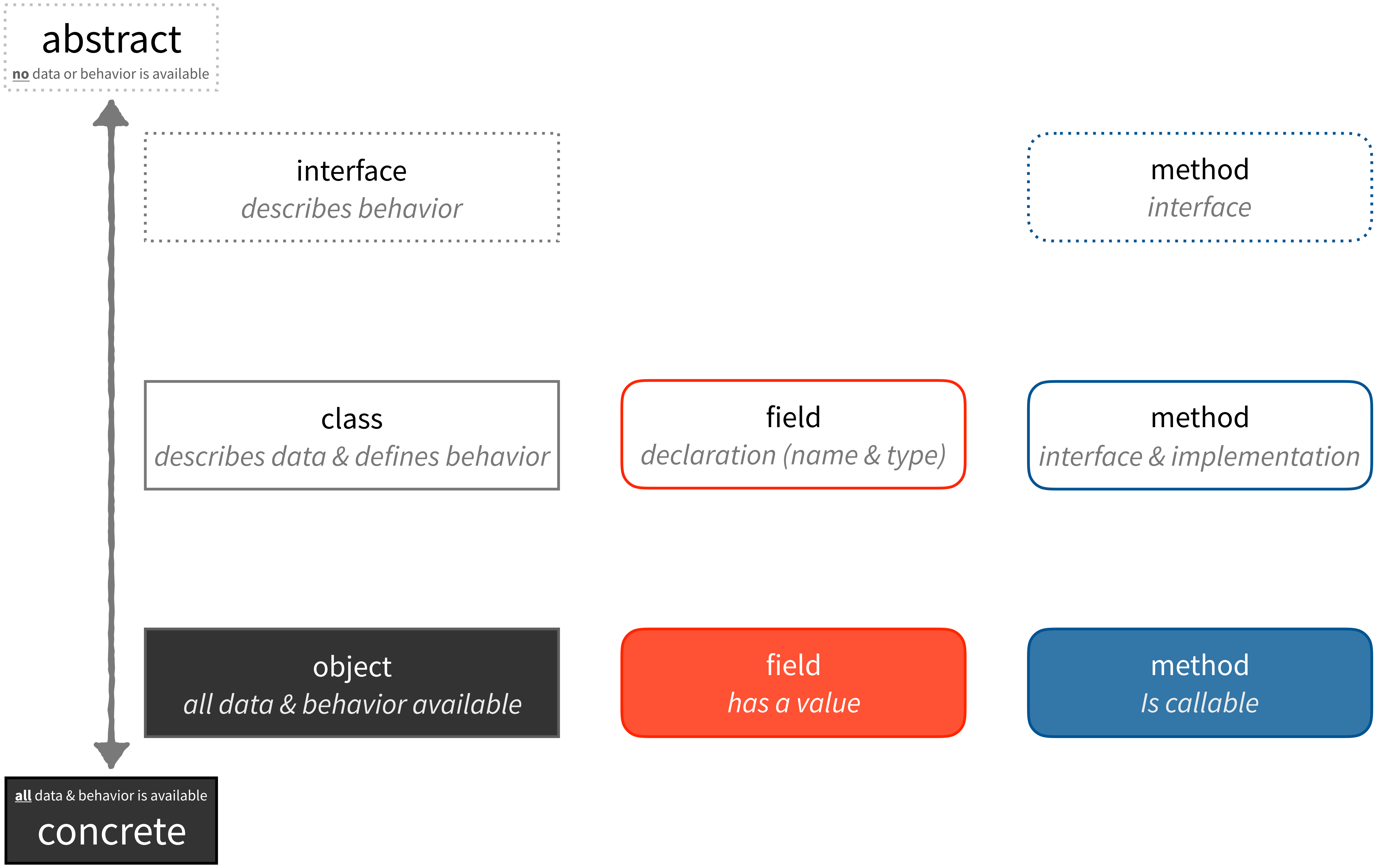
Data + Operations

field declarations   method definitions

- A class describes how to make an object.
- We make an object by combining the description from the class with specific values for the fields.

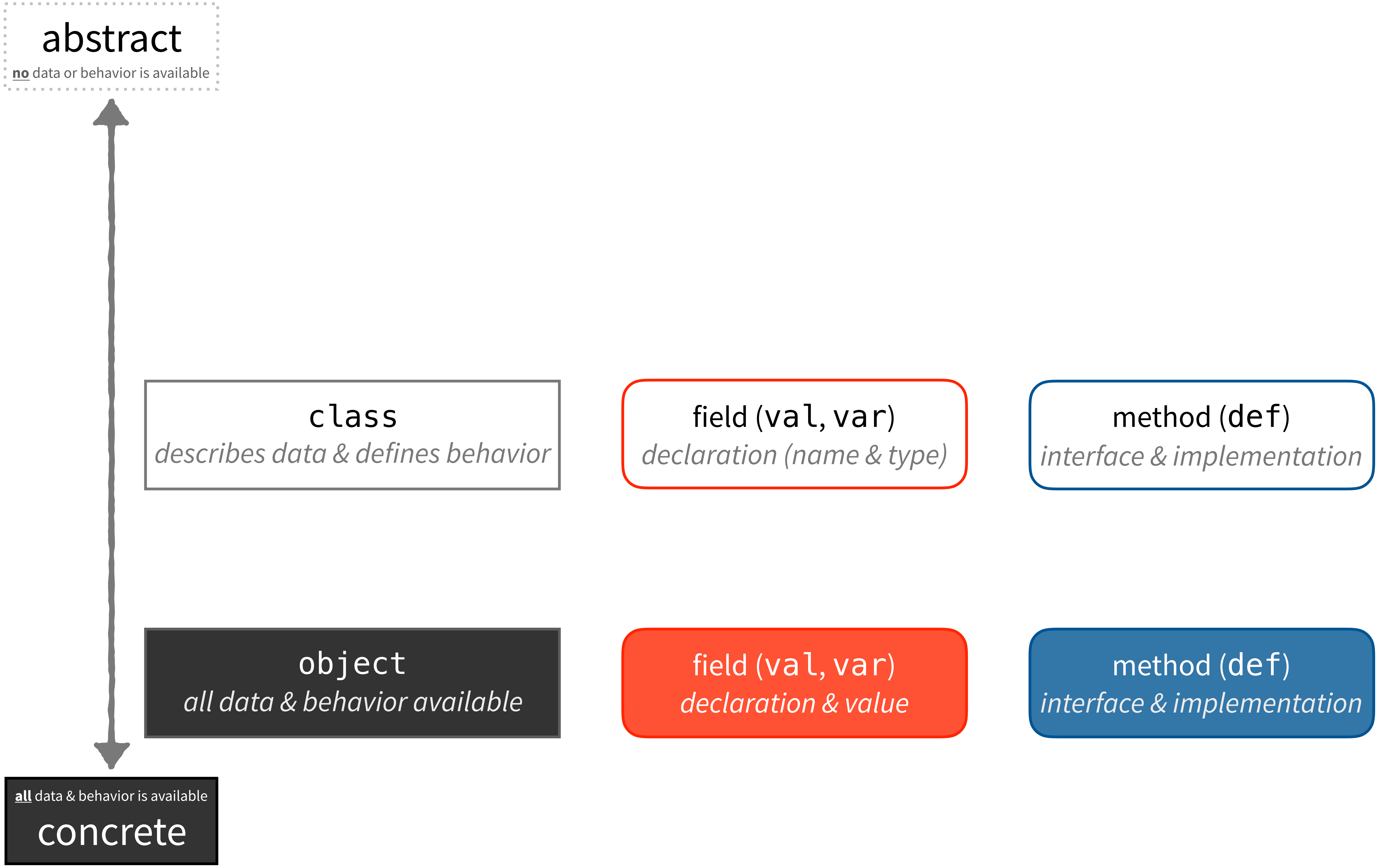
Abstraction in object-oriented programming

disclaimer: This diagram doesn't capture all the nuances of the abstract / concrete spectrum in object-oriented programming. Also, it uses generic terms such as "interface" that may not correspond to terms used by specific languages (e.g., Java).


abstract
no data or behavior is available
interface
describes behavior
method
interface
class
describes data & defines behavior
field
declaration (name & type)
method
interface & implementation
object
all data & behavior available
field
has a value
method
Is callable
all data & behavior is available
concrete

Abstraction in Scala: `classes` & `objects`

disclaimer: This diagram doesn't capture all the nuances of the abstract / concrete spectrum in Scala.


abstract
no data or behavior is available
class
describes data & defines behavior
field (val, var)
declaration (name & type)
method (def)
interface & implementation
object
all data & behavior available
field (val, var)
declaration & value
method (def)
interface & implementation
all data & behavior is available
concrete

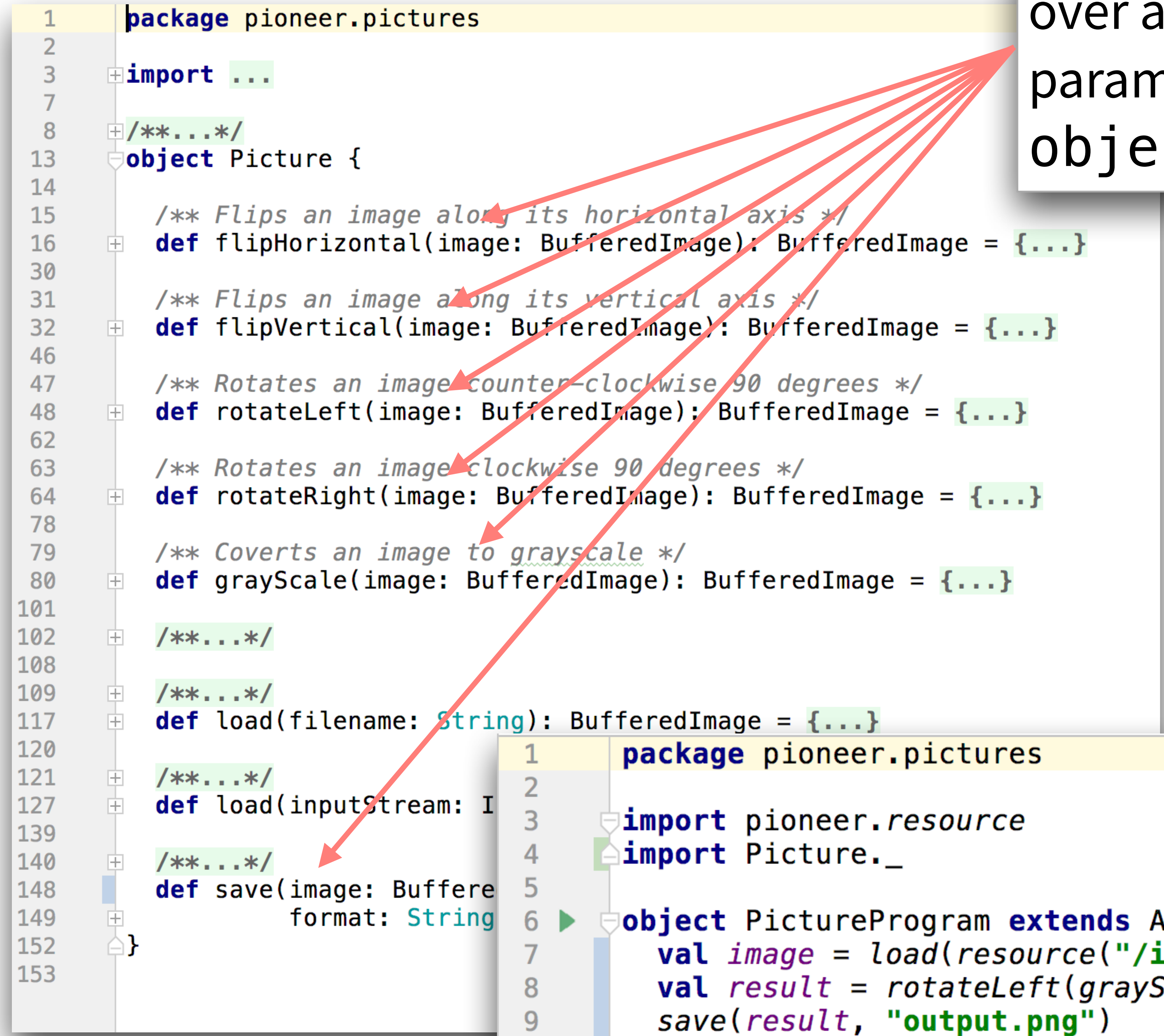
First step: `Picture` objects with methods

```scala
object PictureProgram extends App {
  val image = load(resource("/image.png"))
  val result =
    rotateLeft(
      grayScale(
        flipHorizontal(image)
      )
    )
  save(result, "output0.png")
}
```

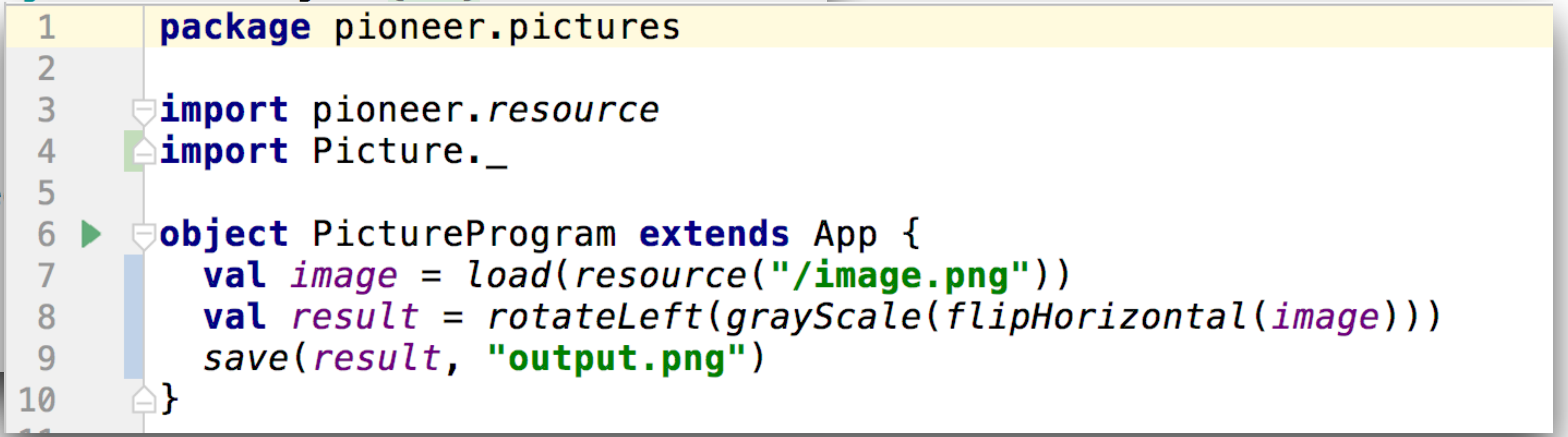
```scala
object PictureProgram extends App {
  val image = load(resource("/image.png"))
  image.flipHorizontal()
  image.grayScale()
  image.rotateLeft()
  image.save("output0.png")
}
```

Currently: functions over parameters

A collection of functions, over a `BufferedImage` parameter, inside an `object`.

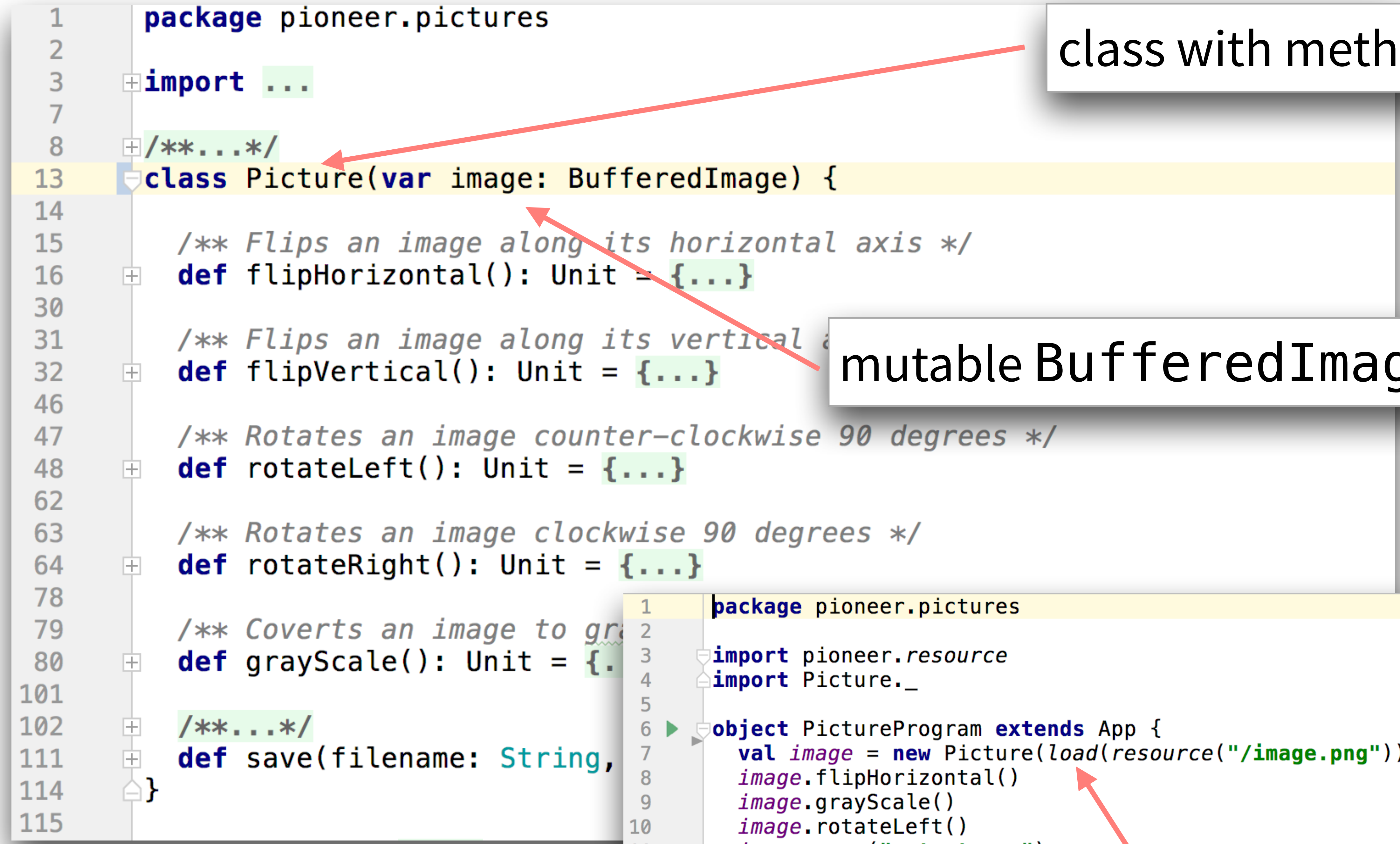


```scala
package pioneer.pictures

import ...

/**...*/
object Picture {

  /** Flips an image along its horizontal axis */
  def flipHorizontal(image: BufferedImage): BufferedImage = {...}

  /** Flips an image along its vertical axis */
  def flipVertical(image: BufferedImage): BufferedImage = {...}

  /** Rotates an image counter-clockwise 90 degrees */
  def rotateLeft(image: BufferedImage): BufferedImage = {...}

  /** Rotates an image clockwise 90 degrees */
  def rotateRight(image: BufferedImage): BufferedImage = {...}

  /** Coverts an image to grayscale */
  def grayScale(image: BufferedImage): BufferedImage = {...}

  /**...*/

  /**...*/
  def load(filename: String): BufferedImage = {...}

  /**...*/
  def load(inputStream: I

  /**...*/
  def save(image: Buffere
           format: String
}
```



```scala
package pioneer.pictures

import pioneer.resource
import Picture._

object PictureProgram extends App {
  val image = load(resource("/image.png"))
  val result = rotateLeft(grayScale(flipHorizontal(image)))
  save(result, "output.png")
}
```

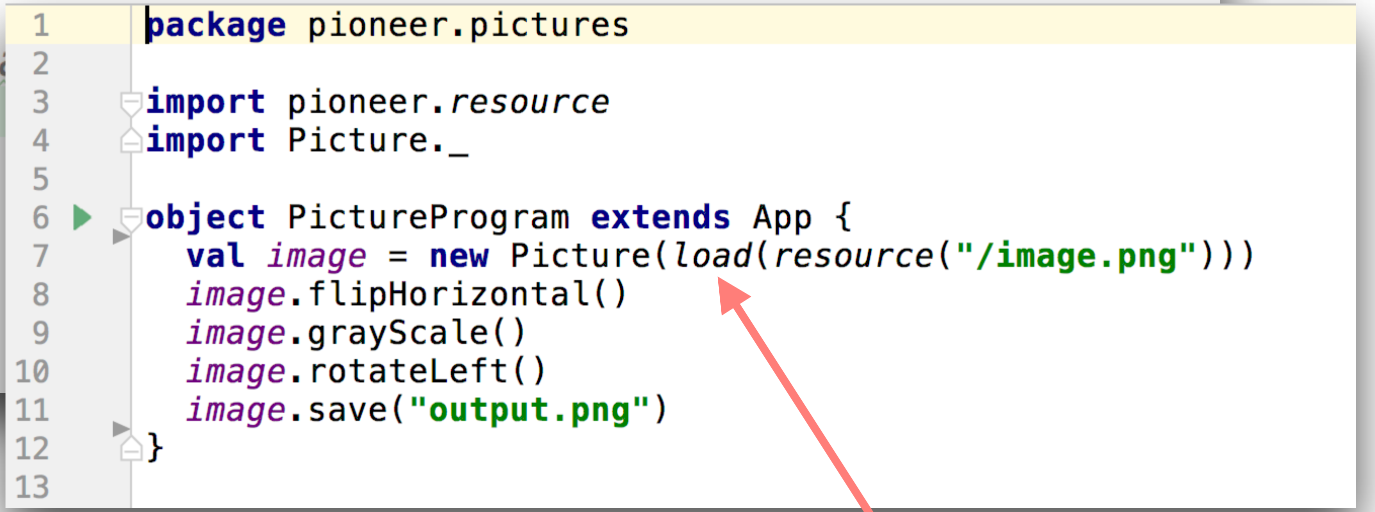
OOP: methods over fields



```
package pioneer.pictures

import ...

/**...*/
class Picture(var image: BufferedImage) {

  /** Flips an image along its horizontal axis */
  def flipHorizontal(): Unit = {...}

  /** Flips an image along its vertical a
  def flipVertical(): Unit = {...}

  /** Rotates an image counter-clockwise 90 degrees */
  def rotateLeft(): Unit = {...}

  /** Rotates an image clockwise 90 degrees */
  def rotateRight(): Unit = {...}

  /** Coverts an image to gr
  def grayScale(): Unit = {.

  /**...*/
  def save(filename: String,
}
```

class with methods

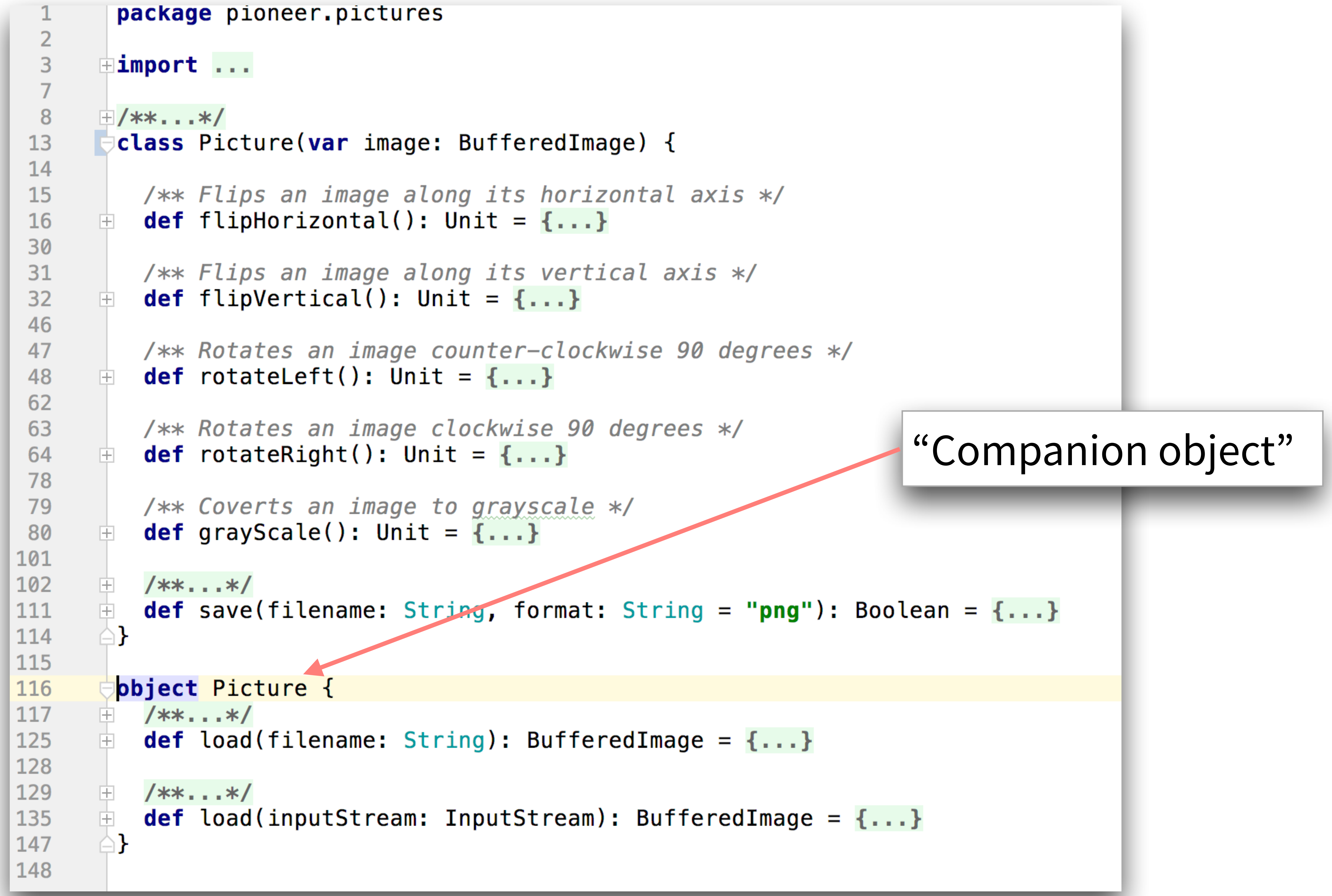
mutable BufferedImage field



```
package pioneer.pictures

import pioneer.resource
import Picture._

object PictureProgram extends App {
  val image = new Picture(load(resource("/image.png")))
  image.flipHorizontal()
  image.grayScale()
  image.rotateLeft()
  image.save("output.png")
}
```

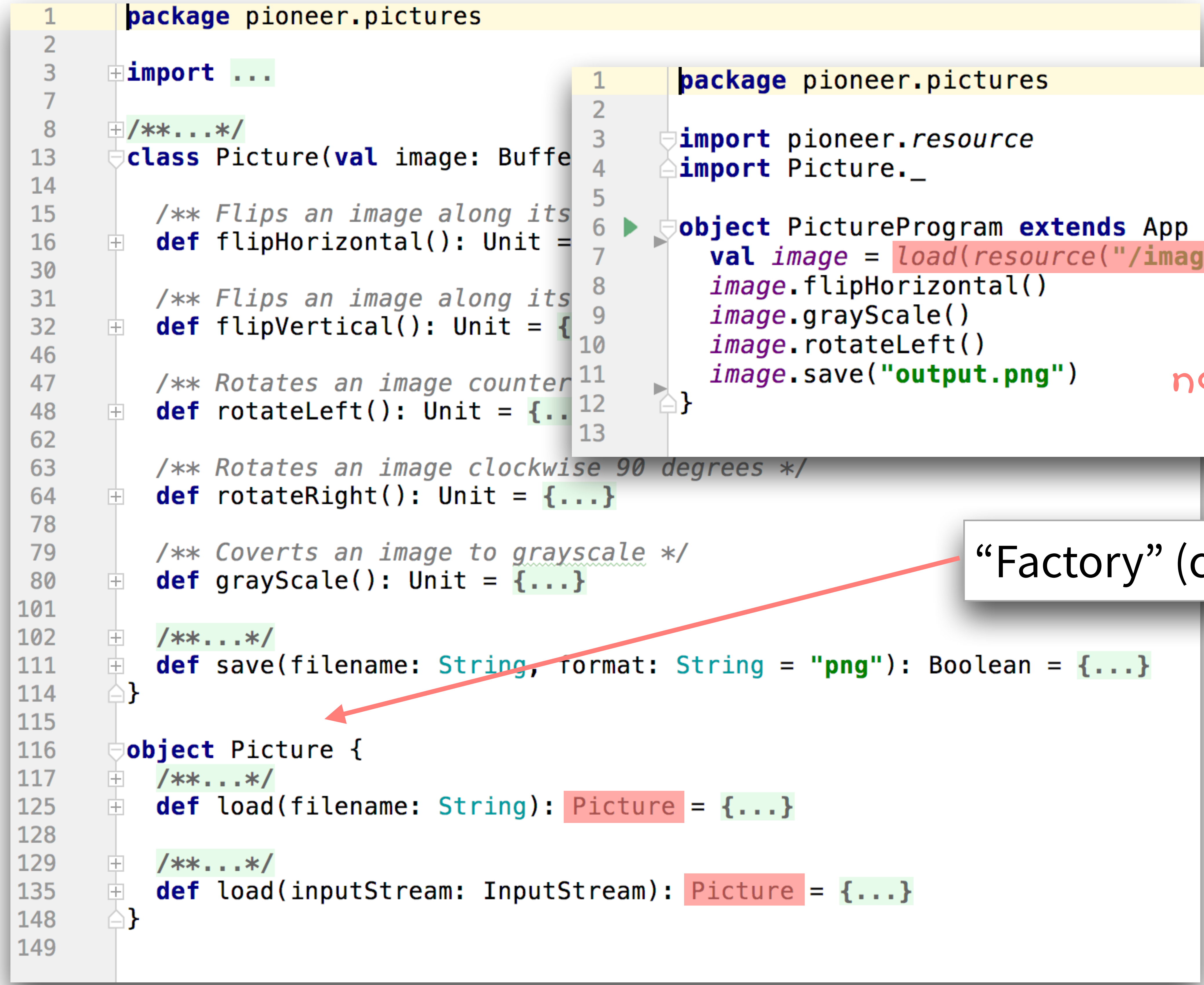
What about load?!

OOP: methods over fields

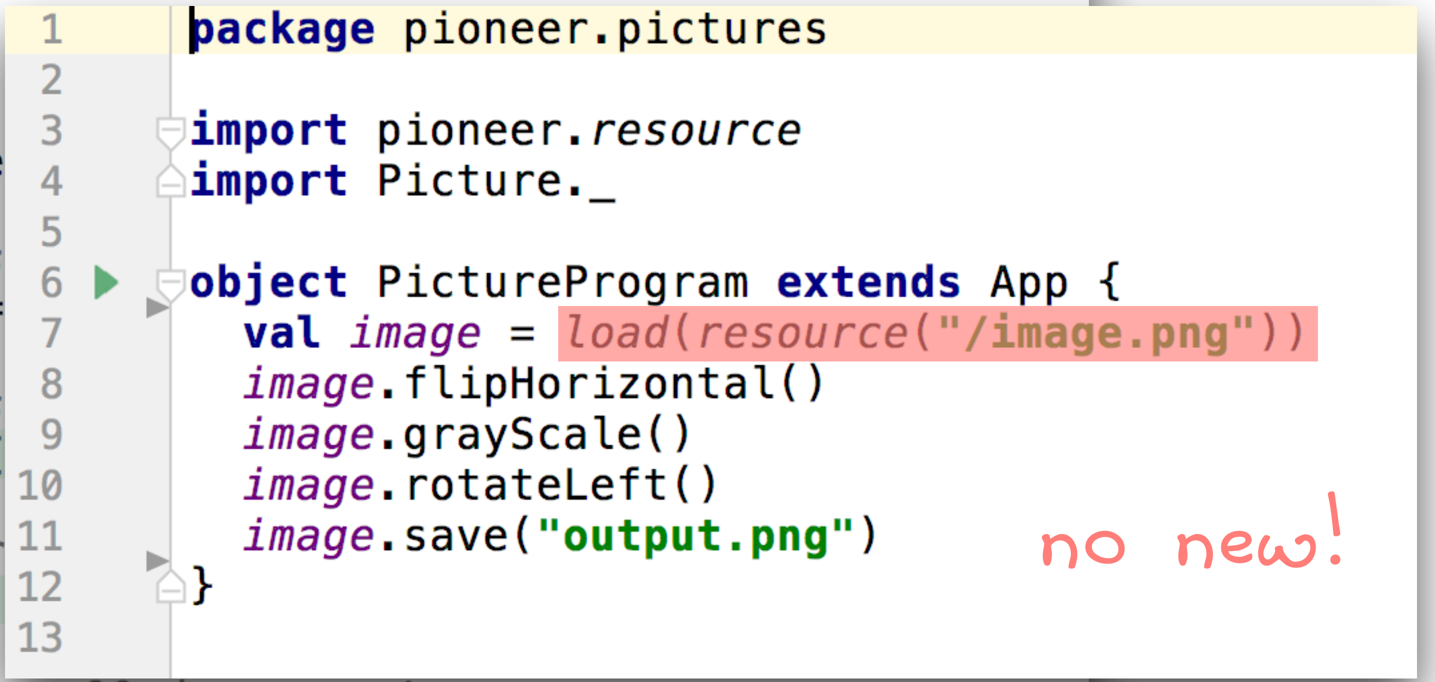


```
package pioneer.pictures

import ...

/**...*/
class Picture(var image: BufferedImage) {

  /** Flips an image along its horizontal axis */
  def flipHorizontal(): Unit = {...}

  /** Flips an image along its vertical axis */
  def flipVertical(): Unit = {...}

  /** Rotates an image counter-clockwise 90 degrees */
  def rotateLeft(): Unit = {...}

  /** Rotates an image clockwise 90 degrees */
  def rotateRight(): Unit = {...}

  /** Coverts an image to grayscale */
  def grayScale(): Unit = {...}

  /**...*/
  def save(filename: String, format: String = "png"): Boolean = {...}
}

object Picture {
  /**...*/
  def load(filename: String): BufferedImage = {...}

  /**...*/
  def load(inputStream: InputStream): BufferedImage = {...}
}
```

Let's make our users a little happier



```
package pioneer.pictures

import ...

/**...*/
class Picture(val image: Buffe

  /** Flips an image along its
  def flipHorizontal(): Unit =

  /** Flips an image along its
  def flipVertical(): Unit = {

  /** Rotates an image counter
  def rotateLeft(): Unit = {..

  /** Rotates an image clockwise 90 degrees */
  def rotateRight(): Unit = {...}

  /** Coverts an image to grayscale */
  def grayScale(): Unit = {...}

  /**...*/
  def save(filename: String, format: String = "png"): Boolean = {...}
}

object Picture {
  /**...*/
  def load(filename: String): Picture = {...}

  /**...*/
  def load(inputStream: InputStream): Picture = {...}
}
```



```
package pioneer.pictures

import pioneer.resource
import Picture._

object PictureProgram extends App {
  val image = load(resource("/image.png"))
  image.flipHorizontal()
  image.grayScale()
  image.rotateLeft()
  image.save("output.png")
}
```

no new!

"Factory" (creates objects)

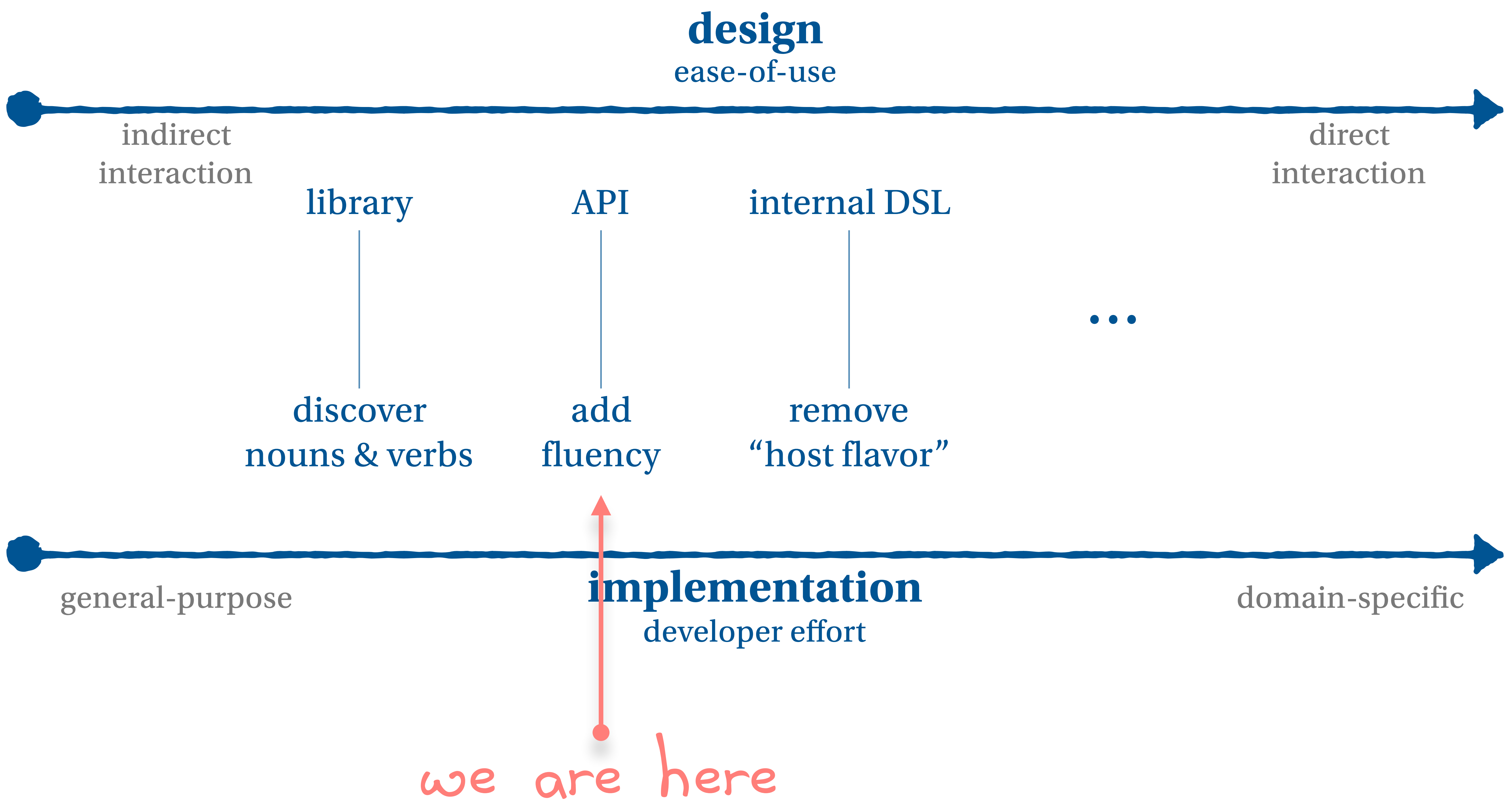
Fluency: chaining in OOP

Chaining is compositional fluency in OOP.

```
object PictureProgram extends App {
  val image = load(resource("/image.png"))
  image.flipHorizontal()
  image.grayScale()
  image.rotateLeft()
  image.save("output0.png")
}
```

```
object PictureProgram extends App {
  val image = load(resource("/image.png"))

  image.flipHorizontal()
       .grayScale()
       .rotateLeft()
       .save("output0.png")
}
```

Implementation techniques



Removing the host-language flavor

How to make it look a little less like Scala?

```
object PictureProgram extends App {
  load(resource("/image.png"))
    .flipHorizontal()
    .grayScale()
    .rotateLeft()
    .save(“output0.png")
}
```

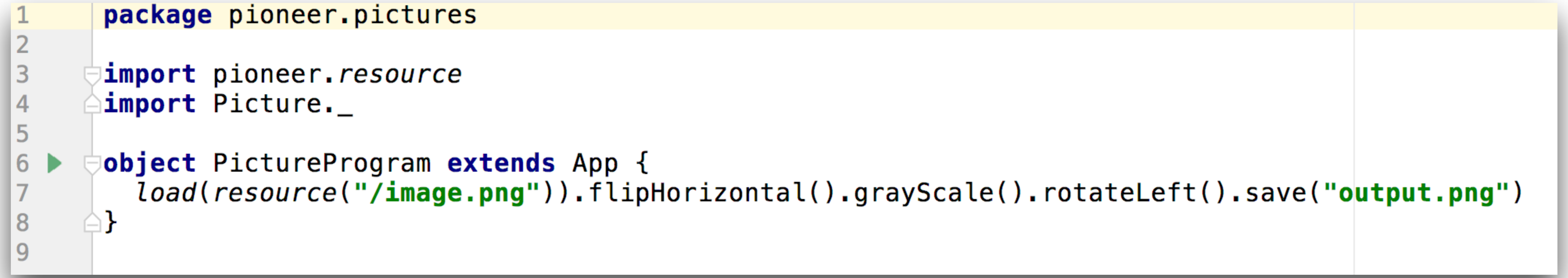
how the library works now

```
object PictureProgram extends App {(
  load(resource("/image.png"))
  flipHorizontal()
  grayScale()
  rotateLeft()
  save("output0.png")
)}
```

how we'd like the library to work

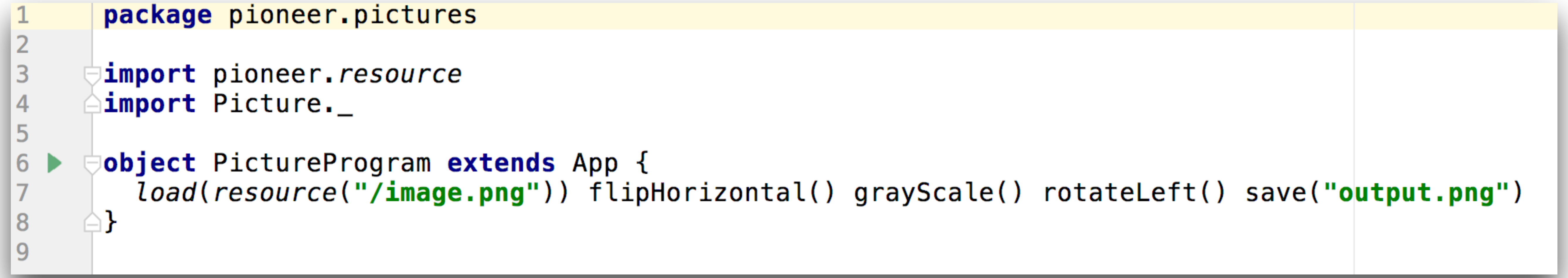
Scala method calls

We can omit the **.** from our method calls.



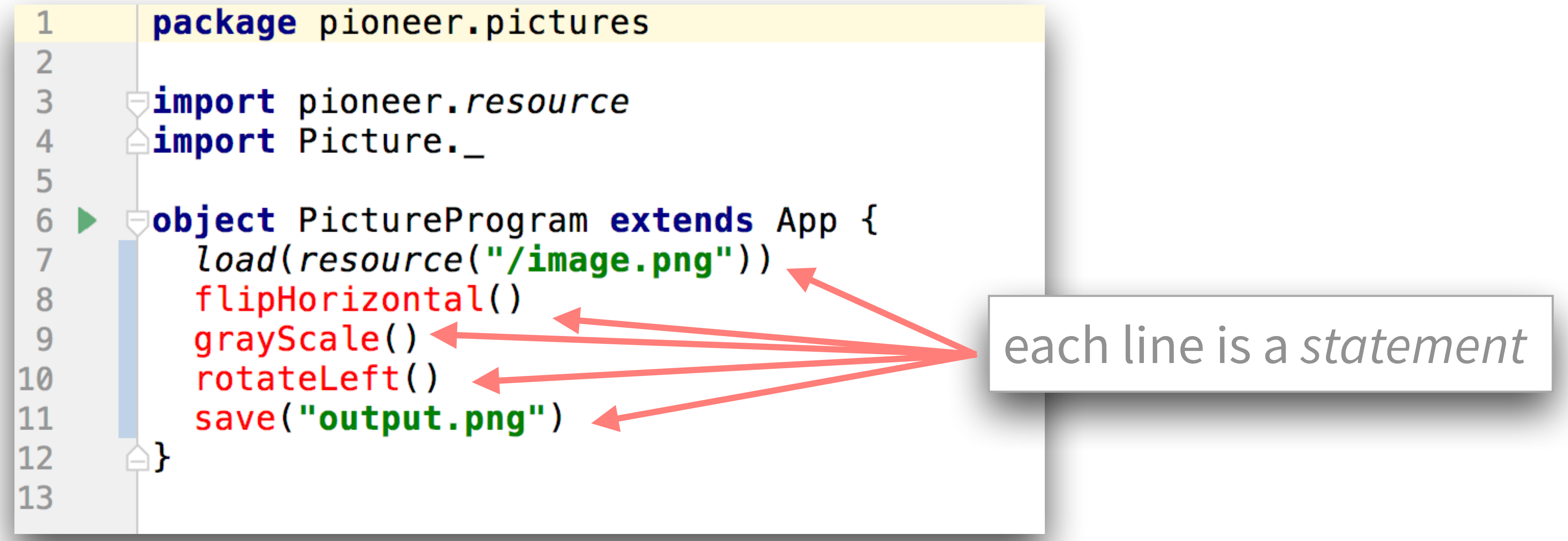
```scala
package pioneer.pictures

import pioneer.resource
import Picture._

object PictureProgram extends App {
  load(resource("/image.png")).flipHorizontal().grayScale().rotateLeft().save("output.png")
}

```

These programs are equivalent!



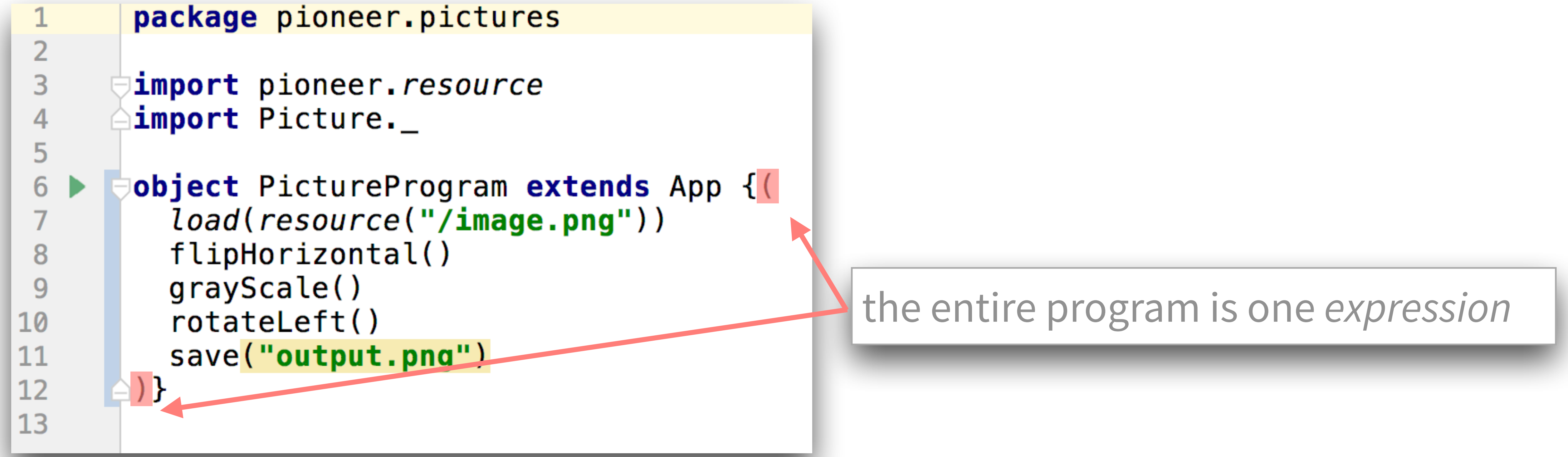
```scala
package pioneer.pictures

import pioneer.resource
import Picture._

object PictureProgram extends App {
  load(resource("/image.png")) flipHorizontal() grayScale() rotateLeft() save("output.png")
}

```

Remember: Scala infers semicolons



```
package pioneer.pictures

import pioneer.resource
import Picture._

object PictureProgram extends App {
  load(resource("/image.png"))
  flipHorizontal()
  grayScale()
  rotateLeft()
  save("output.png")
}
```

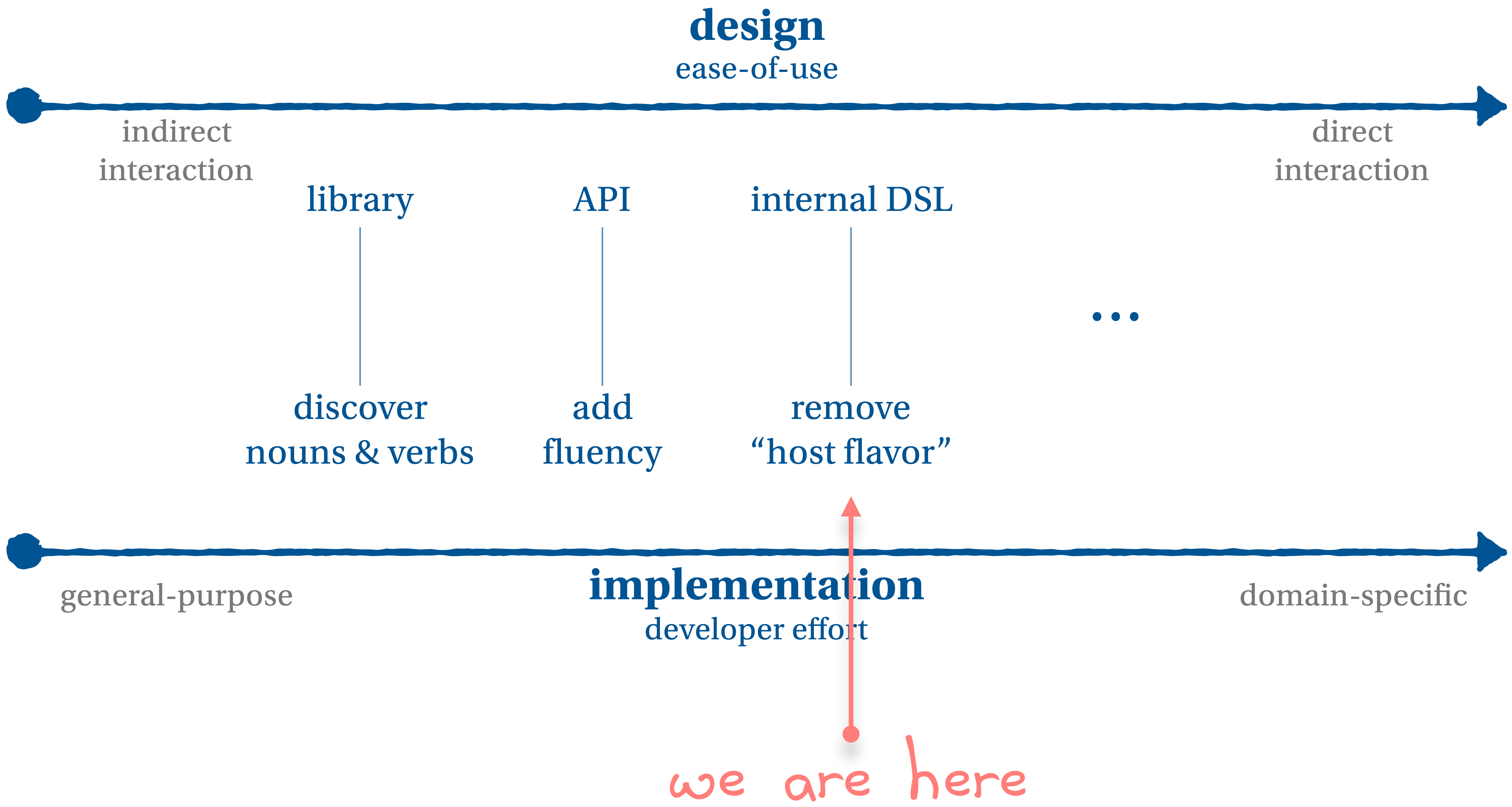
each line is a *statement*



```
package pioneer.pictures

import pioneer.resource
import Picture._

object PictureProgram extends App {(
  load(resource("/image.png"))
  flipHorizontal()
  grayScale()
  rotateLeft()
  save("output.png")
)}
```

the entire program is one *expression*

Implementation techniques


design
ease-of-use
indirect interaction
direct interaction
library
API
internal DSL
…
discover nouns & verbs
add fluency
remove "host flavor"
general-purpose
implementation
developer effort
domain-specific
we are here