Data Abstraction

Principles of Functional Programming

Data Abstraction

The previous example has shown that rational numbers aren't always represented in their simplest form. (Why?)

One would expect the rational numbers to be *simplified*:

- reduce them to their smallest numerator and denominator by dividing both with a divisor.

We could implement this in each rational operation, but it would be easy to forget this division in an operation.

A better alternative consists of simplifying the representation in the class when the objects are constructed:

Rationals with Data Abstraction

```
class Rational(x: Int, y: Int):
  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
  def denom = y / g
  ...
```

`gcd` and `g` are *private* members; we can only access them from inside the `Rational` class.

In this example, we calculate `gcd` immediately, so that its value can be re-used in the calculations of `numer` and `denom`.

Rationals with Data Abstraction (2)

It is also possible to call `gcd` in the code of `numer` and `denom`:

```
class Rational(x: Int, y: Int):
  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
  def numer = x / gcd(x, y)
  def denom = y / gcd(x, y)
```

This can be advantageous if it is expected that the functions `numer` and `denom` are called infrequently.

Rationals with Data Abstraction (3)

It is equally possible to turn `numer` and `denom` into `val`s, so that they are computed only once:

```
class Rational(x: Int, y: Int):
  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
  val numer = x / gcd(x, y)
  val denom = y / gcd(x, y)
```

This can be advantageous if the functions `numer` and `denom` are called often.

The Client's View

Clients observe exactly the same behavior in each case.

This ability to choose different implementations of the data without affecting clients is called *data abstraction*.

It is a cornerstone of software engineering.

Self Reference

On the inside of a class, the name `this` represents the object on which the current method is executed.

Example

Add the functions `less` and `max` to the class `Rational`.

```
class Rational(x: Int, y: Int):

  def less(that: Rational): Boolean =
    numer * that.denom < that.numer * denom

  def max(that: Rational): Rational =
    if this.less(that) then that else this
```

Self Reference (2)

Note that a simple name `m`, which refers to another member of the class, is an abbreviation of `this.m`. Thus, an equivalent way to formulate `less` is as follows.

```
def less(that: Rational): Boolean =
  this.numer * that.denom < that.numer * this.denom
```

Preconditions

Let's say our `Rational` class requires that the denominator is positive.

We can enforce this by calling the `require` function.

```
class Rational(x: Int, y: Int):
  require(y > 0, "denominator must be positive")
  ...
```

`require` is a predefined function.

It takes a condition and an optional message string.

If the condition passed to `require` is `false`, an `IllegalArgumentException` is thrown with the given message string.

Assertions

Besides `require`, there is also `assert`.

Assert also takes a condition and an optional message string as parameters. E.g.

```
val x = sqrt(y)
assert(x >= 0)
```

Like `require`, a failing `assert` will also throw an exception, but it's a different one: `AssertionError` for `assert`, `IllegalArgumentException` for `require`.

This reflects a difference in intent

- `require` is used to enforce a precondition on the caller of a function.
- `assert` is used as to check the code of the function itself.

Constructors

In Scala, a class implicitly introduces a constructor. This one is called the *primary constructor* of the class.

The primary constructor

- takes the parameters of the class
- and executes all statements in the class body (such as the `require` a couple of slides back).

Auxiliary Constructors

Scala also allows the declaration of *auxiliary constructors*.

These are methods named `this`

Example Adding an auxiliary constructor to the class `Rational`.

```
class Rational(x: Int, y: Int):
  def this(x: Int) = this(x, 1)
  ...
```

Rational(2) > *2/1*

End Markers

With longer lists of definitions and deep nesting, it's sometimes hard to see where a class or other construct ends.

End markers are a tool to make this explicit.

```
class Rational(x: Int, y: Int):
  def this(x: Int) = this(x, 1)

  ...
end Rational
```

- And end marker is followed by the name that's defined in the definition that ends at this point.
- It must align with the opening keyword (`class` in this case).

End Markers

End markers are also allowed for other constructs.

```
def sqrt(x: Double): Double =
  ...
end sqrt

if x >= 0 then
  ...
else
  ...
end if
```

If the end marker terminates a control expression such as `if`, the beginning keyword is repeated.

Exercise

Modify the `Rational` class so that rational numbers are kept unsimplified internally, but the simplification is applied when numbers are converted to strings.

Do clients observe the same behavior when interacting with the rational class?

0 yes

0 no

0 yes for small sizes of denominators and nominators and small numbers of operations.