What's the difference between abstraction and generalization?

Asked 8 years, 5 months ago Active 1 year, 11 months ago Viewed 21k times

42

19

I understand that abstraction is about taking something more concrete and making it more abstract. That something may be either a data structure or a procedure. For example:

1. **Data abstraction:** A rectangle is an abstraction of a square. It concentrates on the fact a square has two pairs of opposite sides and it ignores the fact that adjacent sides of a square are equal.
2. **Procedural abstraction:** The higher order function `map` is an abstraction of a procedure which performs some set of operations on a list of values to produce an entirely new list of values. It concentrates on the fact that the procedure loops through every item of the list in order to produce a new list and ignores the actual operations performed on every item of the list.

So my question is this: how is abstraction any different from generalization? I'm looking for answers primarily related to functional programming. However if there are parallels in object-oriented programming then I would like to learn about those as well.

oop functional-programming abstraction nomenclature generalization

Share Improve this question Follow

asked Oct 10, 2013 at 9:19
Aadit M Shah
70k ●27 ●158 ●281

4 Mathematically speaking, abstraction is the existential quantifier and generalization is the universal quantifier. – lukstafi Oct 10, 2013 at 21:02

Report this ad

8 Answers

Sorted by: Date created (oldest first)

46

+100

A very interesting question indeed. I found this article on the topic, which concisely states that:

> While abstraction reduces complexity by hiding irrelevant detail, generalization reduces complexity by replacing multiple entities which perform similar functions with a single construct.

Lets take the old example of a system that manages books for a library. A book has tons of properties (number of pages, weight, font size(s), cover,...) but for the purpose of our library we may only need

```
Book(title, ISBN, borrowed)
```

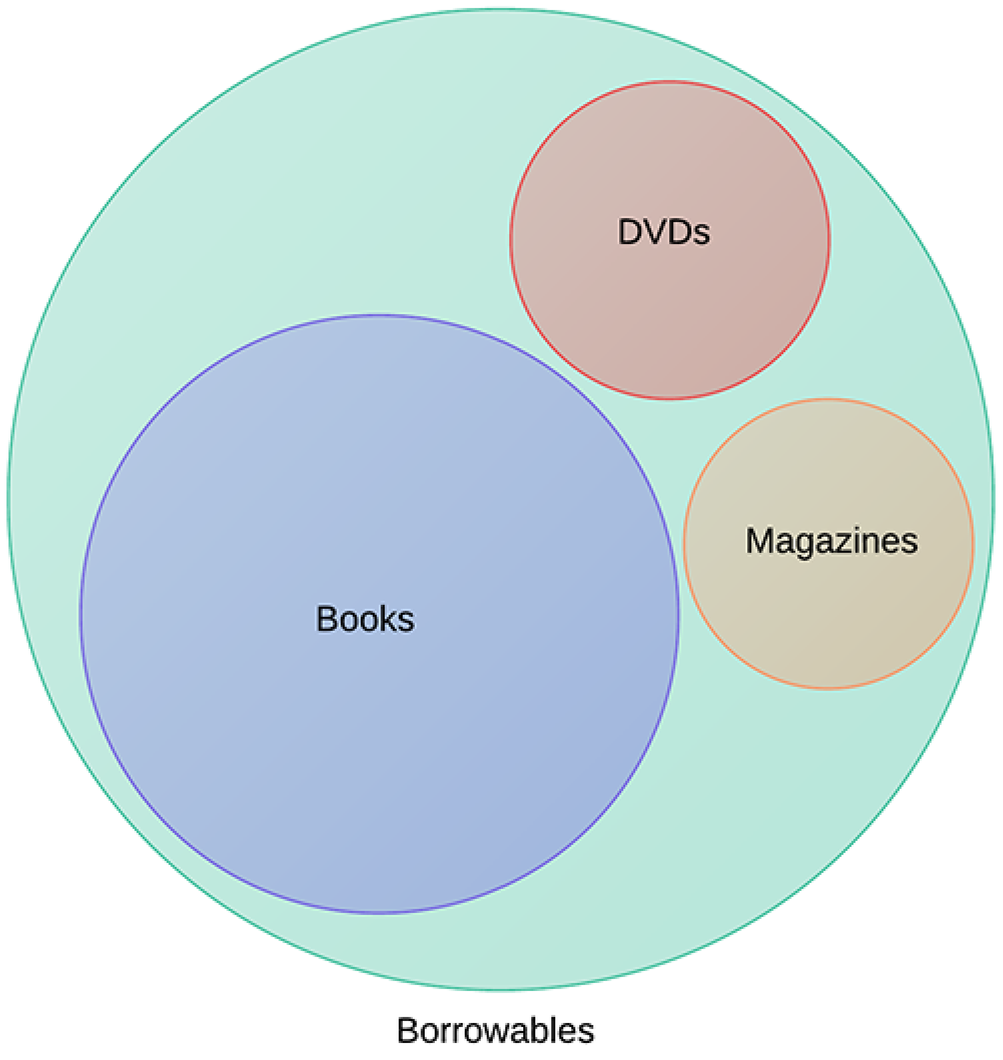
We just abstracted from the real books in our library, and only took the properties that interested us in the context of our application.

Generalization on the other hand does not try to remove detail but to make functionality applicable to a wider (more generic) range of items. Generic containers are a very good example for that mindset: You wouldn't want to write an implementation of `StringList`, `IntList`, and so on, which is why you'd rather write a *generic* List which applies to all types (like `List[T]` in Scala). Note that you haven't *abstracted* the list, because you didn't remove any details or operations, you just made them generically applicable to all your types.

Round 2

@dtldarek's answer is really a very good illustration! Based on it, here's some code that might provide further clarification.

Remeber the `Book` I mentioned? Of course there are other things in a library that one can borrow (I'll call the set of all those objects `Borrowable` even though that probably isn't even a word :D):



All of these items will have an **abstract** representation in our database and business logic, probably similar to that of our `Book`. Additionally, we might define a trait that is common to all `Borrowable`s:

```
trait Borrowable {
    def itemId:Long
}
```

We could then write **generalized** logic that applies to all `Borrowable`s (at that point we don't care if its a book or a magazine):

```
object Library {
    def lend(b:Borrowable, c:Customer):Receipt = ...
    [...]
}
```

To summarize: We stored an **abstract representation** of all the books, magazines and DVDs in our database, because an exact representation is neither feasible nor necessary. We then went ahead and said

> It doesn't matter whether a book, a magazine or a DVD is borrowed by a customer. It's always the same process.

Thus we **generalized** the operation of borrowing an item, by defining all things that one can borrow as `Borrowable` s.

Share Improve this answer Follow

edited Sep 7, 2019 at 11:30
Glorfindel
20.6k ● 13 ● 73 ● 97

answered Oct 10, 2013 at 11:16
fresskoma
24.8k ● 10 ● 82 ● 124

Thank you for your input. It cleared some of the doubts I had about generalization. Nevertheless my understanding of abstraction vs generalization is still a little fuzzy. Perhaps you could provide some code to explain the difference? – Aadit M Shah Oct 13, 2013 at 13:41

1 @AaditMShah Gave it another shot, hope it helps :) – fresskoma Oct 19, 2013 at 11:35

1 Thank you. It certainly did help. I started an additional bounty to reward your answer and I'll award it to you in a week so that your answer gets the most publicity. – Aadit M Shah Oct 20, 2013 at 3:11

2 Hey fresskoma, just revisiting old concepts. So, to be clear abstraction is equivalent to ad-hoc polymorphism and generalization is equivalent to parametric polymorphism. Is that correct? – Aadit M Shah Jul 13, 2019 at 5:33

6

I'm going to use some examples to describe generalisation and abstraction, and I'm going to refer to this article.

To my knowledge, there is no official source for the definition of abstraction and generalisation in the programming domain (Wikipedia is probably the closest you'll get to an official definition in my opinion), so I've instead used an article which I deem credible.

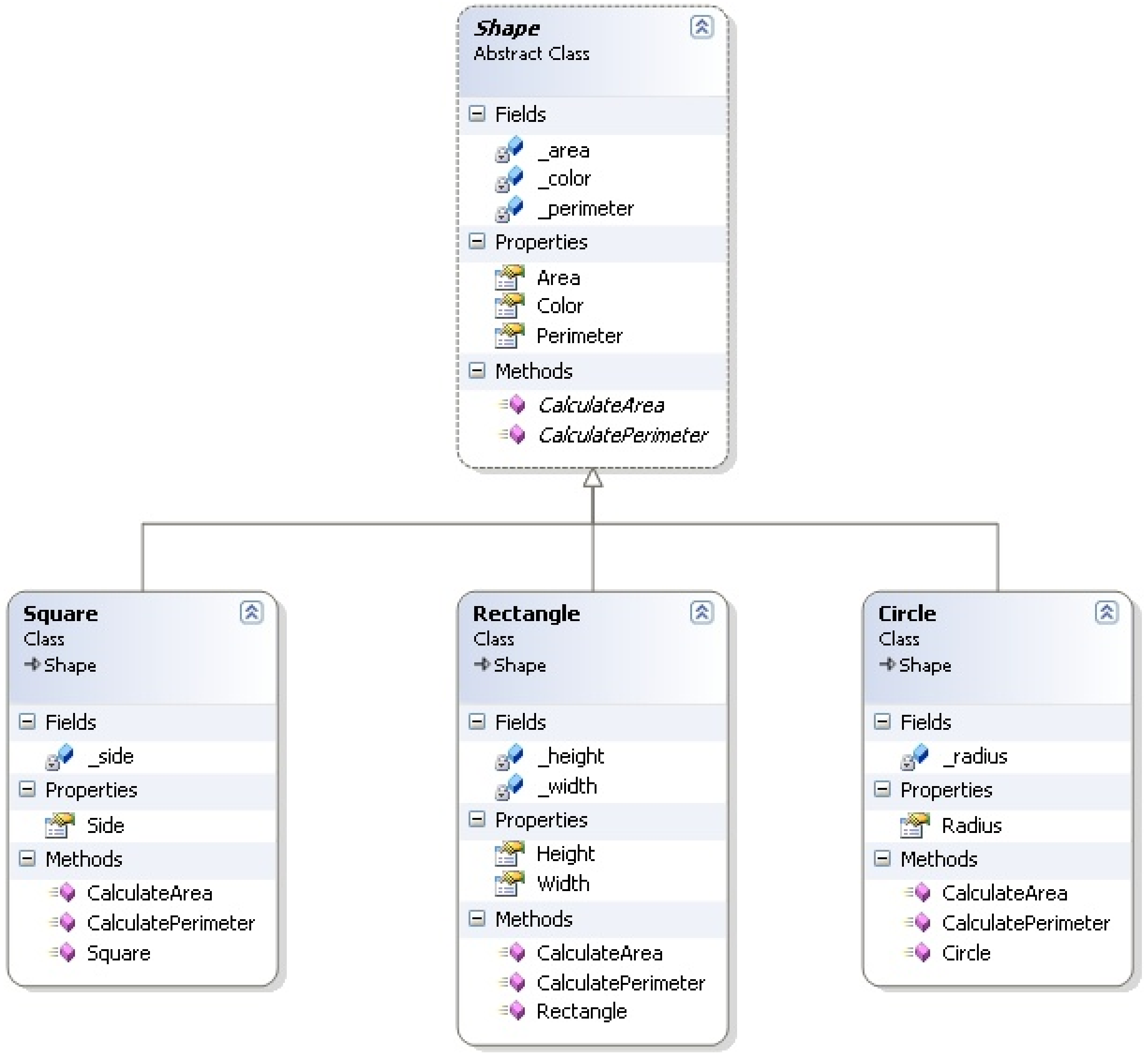
Generalization

The article states that:

> "The concept of generalization in OOP means that an object encapsulates common state and behavior for a category of objects."

So for example, if you apply generalisation to shapes, then the common properties for all types of shape are area and perimeter.

Hence a generalised shape (e.g. Shape) and specialisations of it (e.g. a Circle), can be represented in classes as follows (note that this image has been taken from the aforementioned article)



Similarly, if you were working in the domain of jet aircraft, you could have a Jet as a generalisation, which would have a wingspan property. A specialisation of a Jet could be a FighterJet, which would inherit the wingspan property and would have its own property unique to fighter jets e.g. NumberOfMissiles.

Abstraction

The article defines abstraction as:

> "the process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use" (Richard Gabriel)"

In the domain of programming:

> An abstract class is a parent class that allows inheritance but can never be instantiated.

Hence in the example given in the Generalization section above, a Shape is abstract as:

> In the real world, you never calculate the area or perimeter of a generic shape, you must know what kind of geometric shape you have because each shape (eg. square, circle, rectangle, etc.) has its own area and perimeter formulas.

However, as well as being abstract a shape *is also a generalisation* (because it "encapsulates common state and behavior for a category of objects" where in this case the objects are shapes).

Going back to the example I gave about Jets and FighterJets, a Jet is not abstract as a concrete instance of a Jet is feasible, as one can exist in the real world, unlike a shape i.e. in the real world you cant hold a shape you hold an instance of a shape e.g. a cube. So in the aircraft example, a Jet is not abstract, it is a generalisation as it is possible to have a "concrete" instance of a jet.

Share Improve this answer Follow edited Mar 16, 2015 at 16:26 answered Oct 13, 2013 at 20:23

Ben Smith
17.9k ● 5 ● 59 ● 85

2

Not addressing credible / official source: an example in Scala

Having "Abstraction"

```
trait AbstractContainer[E] { val value: E }

object StringContainer extends AbstractContainer[String] {
  val value: String = "Unflexible"
}

class IntContainer(val value: Int = 6) extends AbstractContainer[Int]

val stringContainer = new AbstractContainer[String] {
  val value = "Any string"
}
```

and "Generalization"

```
def specialized(c: StringContainer.type) =
  println("It's a StringContainer: " + c.value)

def slightlyGeneralized(s: AbstractContainer[String]) =
  println("It's a String container: " + s.value)

import scala.reflect.{ classTag, ClassTag }
def generalized[E: ClassTag](a: AbstractContainer[E]) =
  println(s"It's a ${classTag[E].toString()} container: ${a.value}")

import scala.language.reflectiveCalls
def evenMoreGeneral(d: { def detail: Any }) =
  println("It's something detailed: " + d.detail)
```

executing

```
  specialized(StringContainer)
  slightlyGeneralized(stringContainer)
  generalized(new IntContainer(12))
  evenMoreGeneral(new { val detail = 3.141 })
```

leads to

```
It's a StringContainer: Unflexible
It's a String container: Any string
It's a Int container: 12
It's something detailed: 3.141
```

Share Improve this answer Follow

answered Oct 18, 2013 at 13:52

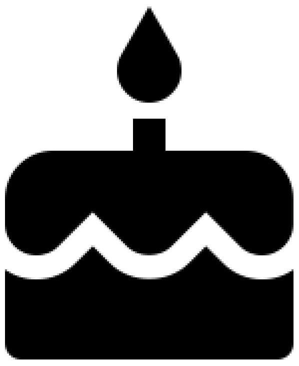
CaringDev
7,881 ● 1 ● 22 ● 40

40

+50

Object:

Abstraction:



Generalization:

Example in Haskell:

The implementation of the selection sort by using priority queue with three different interfaces:

- an open interface with the queue being implemented as a sorted list,
- an abstracted interface (so the details are hidden behind the layer of abstraction),
- a generalized interface (the details are still visible, but the implementation is more flexible).

```
{-# LANGUAGE RankNTypes #-}

module Main where

import qualified Data.List as List
import qualified Data.Set as Set

{- TYPES: -}

-- PQ new push pop
-- by intention there is no build-in way to tell if the queue is empty
data PriorityQueue q t = PQ (q t) (t -> q t -> q t) (q t -> (t, q t))
-- there is a concrete way for a particular queue, e.g. List.null
type ListPriorityQueue t = PriorityQueue [] t
-- but there is no method in the abstract setting
newtype AbstractPriorityQueue q = APQ (forall t. Ord t => PriorityQueue q t)


{- SOLUTIONS: -}

-- the basic version
list_selection_sort :: ListPriorityQueue t -> [t] -> [t]
list_selection_sort (PQ new push pop) list = List.unfoldr mypop (List.foldr
push new list)
  where
    mypop [] = Nothing -- this is possible because we know that the queue is
represented by a list
    mypop ls = Just (pop ls)


-- here we abstract the queue, so we need to keep the queue size ourselves
abstract_selection_sort :: Ord t => AbstractPriorityQueue q -> [t] -> [t]
abstract_selection_sort (APQ (PQ new push pop)) list = List.unfoldr mypop
```

```haskell
(List.foldr mypush (0,new) list)
  where
    mypush t (n, q) = (n+1, push t q)
    mypop (0, q) = Nothing
    mypop (n, q) = let (t, q') = pop q in Just (t, (n-1, q'))


-- here we generalize the first solution to all the queues that allow checking
if the queue is empty
class EmptyCheckable q where
  is_empty :: q -> Bool

generalized_selection_sort :: EmptyCheckable (q t) => PriorityQueue q t -> [t]
-> [t]
generalized_selection_sort (PQ new push pop) list = List.unfoldr mypop
(List.foldr push new list)
  where
    mypop q | is_empty q = Nothing
    mypop q | otherwise  = Just (pop q)


{- EXAMPLES: -}

-- priority queue based on lists
priority_queue_1 :: Ord t => ListPriorityQueue t
priority_queue_1 = PQ [] List.insert (\ls -> (head ls, tail ls))
instance EmptyCheckable [t] where
  is_empty = List.null

-- priority queue based on sets
priority_queue_2 :: Ord t => PriorityQueue Set.Set t
priority_queue_2 = PQ Set.empty Set.insert Set.deleteFindMin
instance EmptyCheckable (Set.Set t) where
  is_empty = Set.null

-- an arbitrary type and a queue specially designed for it
data ABC = A | B | C deriving (Eq, Ord, Show)

-- priority queue based on counting
data PQ3 t = PQ3 Integer Integer Integer
priority_queue_3 :: PriorityQueue PQ3 ABC
priority_queue_3 = PQ new push pop
  where
    new = (PQ3 0 0 0)
    push A (PQ3 a b c) = (PQ3 (a+1) b c)
    push B (PQ3 a b c) = (PQ3 a (b+1) c)
    push C (PQ3 a b c) = (PQ3 a b (c+1))
    pop (PQ3 0 0 0) = undefined
    pop (PQ3 0 0 c) = (C, (PQ3 0 0 (c-1)))
    pop (PQ3 0 b c) = (B, (PQ3 0 (b-1) c))
    pop (PQ3 a b c) = (A, (PQ3 (a-1) b c))

instance EmptyCheckable (PQ3 t) where
  is_empty (PQ3 0 0 0) = True
  is_empty _ = False


{- MAIN: -}

main :: IO ()
main = do
  print $ list_selection_sort priority_queue_1 [2, 3, 1]
  -- print $ list_selection_sort priority_queue_2 [2, 3, 1] -- fail
  -- print $ list_selection_sort priority_queue_3 [B, C, A] -- fail
  print $ abstract_selection_sort (APQ priority_queue_1) [B, C, A] -- APQ hides
the queue
  print $ abstract_selection_sort (APQ priority_queue_2) [B, C, A] -- behind
the layer of abstraction
  -- print $ abstract_selection_sort (APQ priority_queue_3) [B, C, A] -- fail
  print $ generalized_selection_sort priority_queue_1 [2, 3, 1]
  print $ generalized_selection_sort priority_queue_2 [B, C, A]
  print $ generalized_selection_sort priority_queue_3 [B, C, A]-- power of
generalization

  -- fail
  -- print $ let f q = (list_selection_sort q [2,3,1], list_selection_sort q
[B,C,A])
  --         in f priority_queue_1

  -- power of abstraction (rank-n-types actually, but never mind)
  print $ let f q = (abstract_selection_sort q [2,3,1], abstract_selection_sort
q [B,C,A])
          in f (APQ priority_queue_1)

  -- fail
  -- print $ let f q = (generalized_selection_sort q [2,3,1],
generalized_selection_sort q [B,C,A])
  --         in f priority_queue_1
```

The code is also available via pastebin.

Worth noticing are the existential types. As @lukstafi already pointed out, abstraction is similar to existential quantifier and generalization is similar to universal quantifier. Observe that there is a non-trivial connection between the fact that ∀x.P(x) implies ∃x.P(x) (in a non-empty universe), and that there rarely is a generalization without abstraction (even c++-like overloaded functions form a kind of abstraction in some sense).

Credits: Portal cake by Solo. Dessert table by djttwo. The symbol is the cake icon from material.io.

Share Improve this answer Follow

edited Mar 26, 2020 at 10:27

community wiki
4 revs
dtldarek

@AaditMShah Let me know if the Haskell code is useful, or if I should remove it (the answer is less readable this way). – dtldarek Oct 19, 2013 at 17:51

No, the Haskell code is very useful. People who don't understand it can simply skip over it without losing out on the flow of context. – Aadit M Shah Oct 20, 2013 at 2:56

Why no picture of a class? :^) +1 BTW – Sydwell Oct 25, 2013 at 7:55

@dtldarek So generalisation involves abstraction because you take away the set of common properties of different things while abstracting away all other properties. My question is, can we say aggregation ("has a" relationship) also involves abstraction? Is a class Human with members legs, arms, head, body an abstraction? – Patrik Nusszer Jul 4, 2019 at 10:41

@PatrikNusszer I'm not sure if I understand your question. Suppose you want to create a function that does

something useful and works for different types of objects. For that you need all these types to provide you a common API, e.g., each type T provides `.size` that returns a number. Then, even if there is no abstract class or type that specifies the abstract `.size`, that common API is the (potentially implicit and unspecified, but still real) abstraction that your generalization uses. Does that answer your question?
– dtldarek Jul 4, 2019 at 15:18

1

Let me explain in the simplest manner possible.

"All pretty girls are female." is an abstraction.

"All pretty girls put on make-up." is a generalization.

Share Improve this answer Follow

answered Oct 19, 2013 at 9:32
sidquanto
305 ● 3 ● 6

0

Abstraction is usually about reducing complexity by eliminating unnecessary details. For example, an abstract class in OOP is a parent class that contains common features of its children but does not specify the exact functionality.

Generalization does not necessarily require to avoid details but rather to have some mechanism to allow for applying the same function to different argument. For instance, polymorphic types in functional programming languages allow you not to bother about the arguments, rather focus on the operation of the function. Similarly, in java you can have generic type which is an "umbrella" to all types while the function is the same.

Share Improve this answer Follow

answered Oct 19, 2013 at 16:31
NeoSer
507 ● 3 ● 14

2

Abstraction

Abstraction is specifying the framework and hiding the implementation level information. Concreteness will be built on top of the abstraction. It gives you a blueprint to follow to while implementing the details. Abstraction reduces the complexity by hiding low level details.

Example: A wire frame model of a car.

Generalization

Generalization uses a “is-a” relationship from a specialization to the generalization class. Common structure and behaviour are used from the specializtion to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a “Is-a” relationship.

Example: Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here the relationship between student and person, similarly faculty and person is generalization.

Share Improve this answer Follow

answered Oct 19, 2013 at 18:49
Cornel Marian
2,313 ● 20 ● 27

2

I'd like to offer an answer for the greatest possible audience, hence I use the Lingua Franca of the web, Javascript.

Let's start with an ordinary piece of imperative code:

```
// some data

const xs = [1,2,3];

// ugly global state

const acc = [];

// apply the algorithm to the data

for (let i = 0; i < xs.length; i++) {
  acc[i] = xs[i] * xs[i];
}

console.log(acc); // yields [1, 4, 9]
```

Run code snippet Expand snippet

In the next step I introduce the most important abstraction in programming - functions. Functions abstract over expressions:

```
// API

const foldr = f => acc => xs => xs.reduceRight((acc, x) => f(x) (acc), acc);
const concat = xs => ys => xs.concat(ys);
const sqr_ = x => [x * x]; // weird square function to keep the example simple

// some data

const xs = [1,2,3];

// applying
```

```
console.log(
  foldr(x => acc => concat(sqr_(x)) (acc)) ([]) (xs) // [1, 4, 9]
)
```

Run code snippet   Expand snippet

As you can see a lot of implementation details are abstracted away. Abstraction means **the suppression of details**.

Another abstraction step...

```
// API

const comp = (f, g) => x => f(g(x));
const foldr = f => acc => xs => xs.reduceRight((acc, x) => f(x) (acc), acc);
const concat = xs => ys => xs.concat(ys);
const sqr_ = x => [x * x];

// some data

const xs = [1,2,3];

// applying

console.log(
  foldr(comp(concat, sqr_)) ([]) (xs) // [1, 4, 9]
);
```

Run code snippet   Expand snippet

And another one:

```
// API

const concatMap = f => foldr(comp(concat, f)) ([]);
const comp = (f, g) => x => f(g(x));
const foldr = f => acc => xs => xs.reduceRight((acc, x) => f(x) (acc), acc);
const concat = xs => ys => xs.concat(ys);
const sqr_ = x => [x * x];

// some data

const xs = [1,2,3];

// applying

console.log(
  concatMap(sqr_) (xs) // [1, 4, 9]
);
```

Run code snippet Expand snippet

The underlying principle should now be clear. I'm still dissatisfied with `concatMap` though, because it only works with `Array`s. I want it to work with every data type that is foldable:

```
// API

const concatMap = foldr => f => foldr(comp(concat, f)) ([]);
const concat = xs => ys => xs.concat(ys);
const sqr_ = x => [x * x];
const comp = (f, g) => x => f(g(x));

// Array

const xs = [1, 2, 3];

const foldr = f => acc => xs => xs.reduceRight((acc, x) => f(x) (acc), acc);

// Option (another foldable data type)

const None =      r => f => r;
const Some = x => r => f => f(x);

const foldOption = f => acc => tx => tx(acc) (x => f(x) (acc));

// applying

console.log(
  concatMap(foldr) (sqr_) (xs), // [1, 4, 9]
  concatMap(foldOption) (sqr_) (Some(3)), // [9]
  concatMap(foldOption) (sqr_) (None) // []
);
```

Run code snippet Expand snippet

I **broadened the application** of `concatMap` to encompass a larger domain of data types, nameley all foldable datatypes. Generalization emphasizes the commonalities between different types, (or rather objects, entities).

I achieved this by means of dictionary passing (`concatMap`'s additional argument in my example). Now it is somewhat annoying to pass these type dicts around throughout your code. Hence the Haskell folks introduced type classes to, ...um, abstract over type dicts:

```
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]

concatMap (\x -> [x * x]) ([1,2,3]) -- yields [1, 4, 9]
concatMap (\x -> [x * x]) (Just 3) -- yields [9]
concatMap (\x -> [x * x]) (Nothing) -- yields []
```

So Haskell's generic `concatMap` benefits from both, abstraction and generalization.

Share Improve this answer Follow

edited Jun 24, 2017 at 21:37

answered Jun 24, 2017 at 13:01

user6445533

Just to confirm, so you think that a function like `doesBrowserSupportTransparentImages()` is an abstraction? Which function isn't then? Isn't it problematic to replace the verbs 'extract method' with 'abstract'? – Izhaki Sep 15, 2017 at 9:50

@Izhaki Yes, every function is an abstraction, regardless of its name. The name is rather an indicator whether you follow the separation of concerns principle properly. When you can describe the semantics of a function with a single or a few words, you are fine. Otherwise you should refactor. – user6445533 Sep 15, 2017 at 12:01