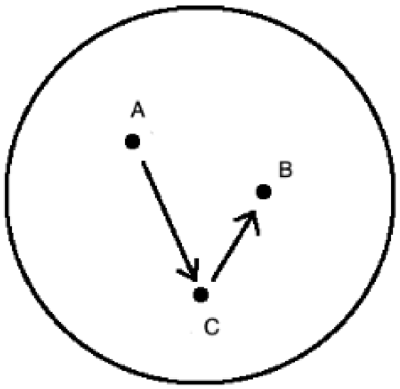
Categories

A **category** is a collection of **objects**, with arrows (also called **morphisms**) between these objects. (There are some other restrictions, but our examples will satisfy them so we'll skip them here.)

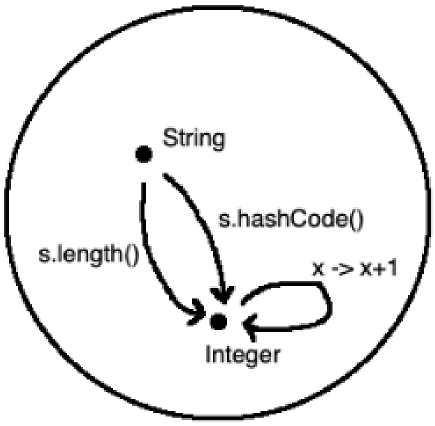


But don't think of the objects as primitive things like numbers; the objects are usually sets. For programmers, there's actually only one practical category to think about: the **category of types**.

In the category of types, the objects are data types, such as `Integer`, `String`, `Point`, `Optional<Integer>`. Each data type is a set of all possible values with that data type.

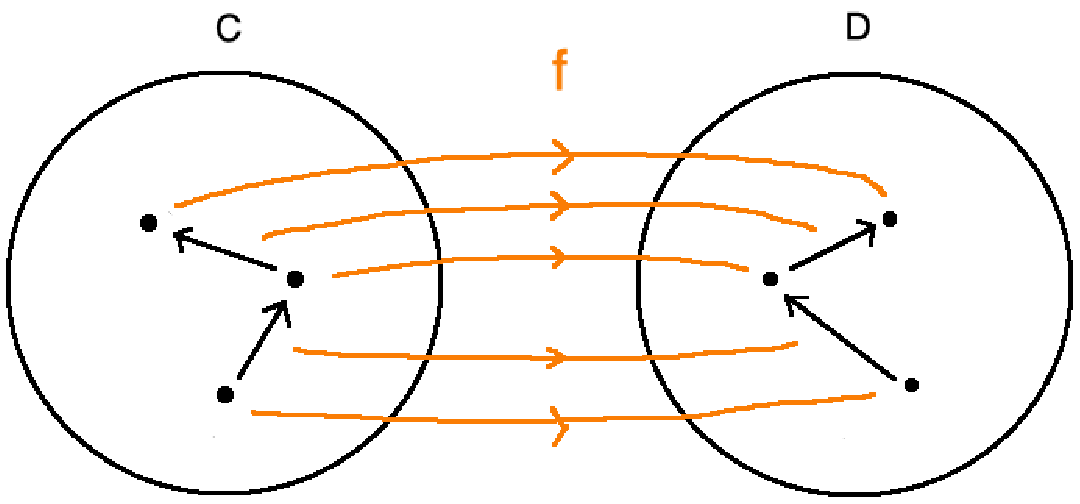
The arrows are (unary) functions that take an input in one data type and produce an output in another data type.

The Category of Types



Functors

Suppose we have two categories, C and D, each with some sets and arrows. A **functor** is a mapping from the sets of C to the sets of D, and from the arrows of C to the arrows of D. One restriction is that an arrow between two sets must map to an arrow between the two mapped sets. (There are some other restrictions, but our examples will satisfy them so we'll skip them here.) So here's an example of a functor f:



But I said there's only one category that we care about - the category of types. So the only functors that we care about are functors from this category to itself. Functors from a category to itself are called **endofunctors**, so an endofunctor of the category of types maps each data type to another data type, and each function to another function.

An example of such a functor is `Optional`. It maps each data type `T` to `Optional<T>`. Now take a function: say `String::length`, which converts a `String` to an `Integer`. The functor will map the function to `x -> x.map(String::length)`, which converts an `Optional<String>` to an `Optional<Integer>`.

Another example of a functor is `List`. It maps each data type `T` to `List<T>`. The function `String::length` will be mapped to the function to `x -> x.map(String::length)`, which converts a `List<String>` to an `List<Integer>`.

So a functor can be thought of as a "wrapper", where the wrapper still knows how to apply arbitrary functions to its contents, usually implemented by a method named `map`.

Applicatives

A functor lets us apply a unary function to a “wrapped” value, but what if we want to apply a binary function to two “wrapped” values? As a concrete example, say we want to implement this:

```
// returns the sum of the two values if both are present,
// otherwise returns empty
Optional<Integer> add(Optional<Integer> a, Optional<Integer> b);
```

If you play around with this, you’ll see that it’s not possible to implement this using only `Optional::map`. The following will return an `Optional<Optional<Integer>>`, not an `Optional<Integer>`:

```
Optional<Integer> add(Optional<Integer> a, Optional<Integer> b) {
    return a.map(aa -> b.map(bb -> aa + bb)); // doesn’t compile!
}
```

For most developers, the most obvious fix is to change the first `map` to a `flatMap`. Supporting `flatMap` will give you a monad, which is the next topic. But it turns out that there’s a weaker thing that we can add to the `Optional` class if we only need to implement our binary function:

```
// method on Optional<T>
// returns the function applied to this value if both this value and
// the function are present, otherwise returns empty
Optional<U> apply(Optional<Function<T, U>> function);
```

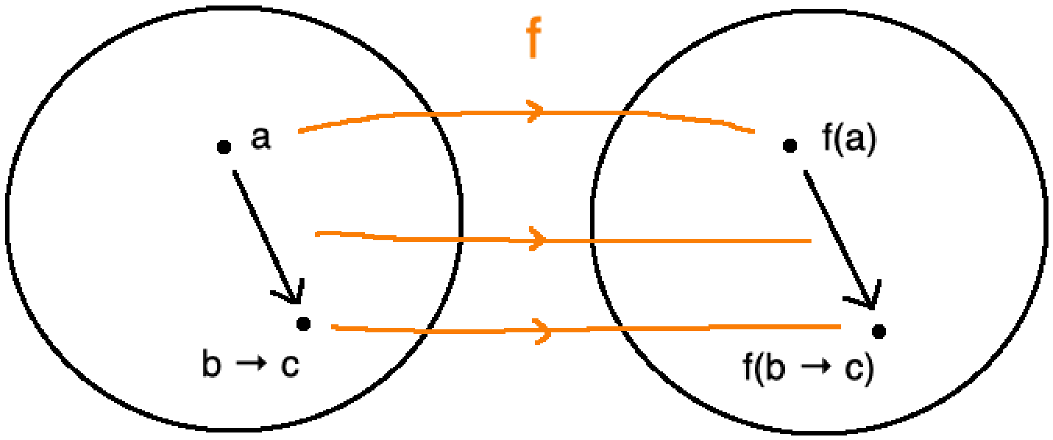
This doesn’t exist in the `Optional` class, probably because `flatMap` is more familiar to developers, and is also more powerful. But if we only have this weaker, hypothetical function `apply` (and `map`), it’s enough to implement our binary function:

```
Optional<Integer> add(Optional<Integer> a, Optional<Integer> b) {
    return a.apply(b.map(bb ->
            (Function<Integer, Integer>) (x -> x + bb)));
}
```

There’s another small thing we need to be able to use `Optional`s: `Optional::of`, to create one in the first place.

So an **applicative functor** is a functor that also supports `of`, which takes a normal value and returns a wrapped value, and `apply`, which takes a wrapped function and a wrapped value, and returns the wrapped result value. An applicative functor lets us apply binary (which can then be extended to ternary, quaternary, etc.) functions to wrapped values.

Here's another way to think about this with a picture. Here, a→(b→c) represents a binary function (by currying, a binary function is equivalent to taking the first value as input and returning a unary function taking only the second value as input). A generic functor f can map this to fa→f(b→c), but in order to get the desired mapped binary function fa→(fb→fc), we need the apply operator f(b→c)→(fb→fc



Monads

A functor knows how to apply unary functions to wrapped values, and an applicative functor knows how to apply binary (and ternary, quaternary, etc.) functions to wrapped values. What can an applicative functor still not do?

Suppose we have a function that takes a normal value and returns a wrapped value. For example, suppose a function `String::validate` returns an `Optional<String>`, which contains the original string if it is valid, and is empty otherwise. But what if we currently only have a wrapped value (maybe from a previous validate function)? Essentially we need to be able to implement this:

```
// returns the function applied to this value if the input value is
// present, otherwise returns empty
Optional<String> flatMap(
        Optional<String> a,
        Function<String, Optional<String>> function);
```

If you play around with this, you'll see that it's not possible to implement this using only `Optional::map` and the additional `Optional::apply` method defined in the previous section.

So an applicative functor cannot necessarily support `flatMap`. However, `Optional` and `List` do support `flatMap`, and that means they are something more specific than applicative functors: they are examples of monads.

A **monad** is a functor that also supports `of`, `apply`, and `flatMap`. And it turns out `apply` can be implemented with `flatMap`, so the only necessary requirements are `of` and `flatMap`. A monad lets us apply functions that return wrapped values.

There's an alternative method that could be added to `Optional`, which turns out to be equivalent in power to `flatMap`:

```
Optional<T> flatten(Optional<Optional<T>> a) {
    return a.flatMap(aa -> aa);
}
```

It’s equivalent because `flatMap` can also be implemented with `flatten`:

```
Optional<T> flatMap(
        Optional<T> a, Function<T, Optional<U>> function) {
    return a.map(function).flatten();
}
```

So an alternative definition of a monad is a functor that also supports `of` and `flatten`.

A monad is a monoid in the category of endofunctors

This is a legendary phrase making fun of the obscureness of category theory terminology in programming. But we almost have enough knowledge to explain this phrase.

We’ll first define the **category of endofunctors**. (I know I had mentioned the category of types was the only practical category. I still stand behind that - explaining the phrase isn’t really a practical exercise after all.)

The objects in this category are the endofunctors: `Optional`, `List`, etc. Unlike data types, endofunctors aren’t “sets of things”, so you’ll have to think of these objects as black boxes.

The arrows in this category are **natural transformations** between two endofunctors. In our scenario, think of a natural transformation as a generic (defined for every data type `T`) function that maps values with one kind of wrapper to values with another kind of wrapper. (There are some other restrictions, but our examples will satisfy them so we’ll skip them here.) For example, this function is a natural transformation from `Optional` to `List`:

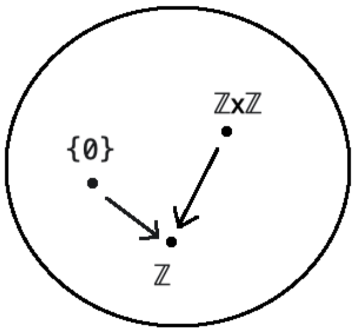
```
List<T> toList(Optional<T> value) {
    return value.stream().toList();
}
```

Now we need to define a monoid. A **monoid** is a set of items, and an operation that you can perform on pairs of items to get another of the same type of items. For example, the integers under addition: you can add two integers to get another integer. Or, the strings under concatenation: you can concatenate two strings to get another string. There must also be an identity element: something that doesn’t affect the other item. For example, 0 is the identity under addition. The empty string is the identity under concatenation.

But endofunctors aren't sets of items, so this definition doesn't make sense for our category of endofunctors. Instead, we need to generalize the definition into category theory language, avoiding anything that requires the objects to be sets.

For integer addition, consider a category where one object consists of the integers ℤ, and another object consists of ordered pairs of integers ℤxℤ. Then there is an arrow between them that represents addition.

We also have an object with a single element, which we'll call 0. An arrow from {0} to ℤ maps 0 to 0 in ℤ, so the arrow "encodes" the identity under addition.



So to generalize this: a monoid is an object T in a category, along with an arrow from the "cross-product object" TxT and an arrow from the "identity object" I. These objects have to be defined in a way that satisfies the category restrictions.

What is a monoid in the category of endofunctors? Well, first we take any endofunctor; let's say `Optional`. The cross-product object is the functor that maps `T` to `Optional<Optional<T>>`, so the corresponding arrow is a natural transformation from `Optional<Optional<T>>` to `Optional<T>`, which is just `flatten`! And the identity object is the functor that maps `T` to `T`, so the corresponding arrow is a natural transformation from `T` to `Optional<T>`, which is just `of`!

So a monoid in the category of endofunctors is an endofunctor with `flatten` and `of` - a monad!

—

Functor: `(a → b) → fa → fb`
Applicative: `f(a → b) → fa → fb`
Monad: `(a → fb) → fa → fb`