
Object
Oriented
Programming
Paradigm

Shifting to OOP

As procedural programming became more and more mainstream, computer scientists started to notice the issue behind states and side effects.

Shifting to OOP

Some languages offered a complete paradigm shift, completely abandoning the notion of state.

This formed the alternative paradigm family, declarative paradigm.

Shifting to OOP

Other languages however, went on the direction of fixing state by introducing richer features and more intuitive design

Shifting to OOP

From this approach the paradigm object-oriented programming was born.

Issues in Maintainability

As programmers started to build bigger and bigger systems, they started to notice the issues in terms of maintainability.

Issues in Maintainability

Hardware became faster and systems became more complex so the concern shifted from **code that runs**, to **code that was intuitive**.

Code that runs

Assembly code was not really built to be intuitive. It was built to reliably work.

Assembly contained mechanisms to move data around and to control program flow.

Code that runs

These features were enough for that time since the goal of programming was to write efficient code that bulky slow CPUs understand.

Code that is intuitive

The landscape started to change when hardware started becoming better and software started becoming more complex.

Code that is intuitive

Speed and memory wasn't that much of an issue anymore so computer scientists' focus shifted towards the issue of maintainability.

Code that is intuitive

Software became bigger and more complex and understanding other programmers' code became more and more difficult.

Code that is intuitive

Because of this writing code that a computer understands isn't enough anymore, a good programmer must write code that humans understand as well.


Code that is intuitive
Code shifted from computer centered to human centered.

Staying in the imperative family

But instead of redesigning the concept of imperative paradigm to solve maintainability, object oriented programming sought to build on top of the features of procedural programming.

Staying in the imperative family

State still exists but OOP gave imperative programmers extra tools to protect code from being carelessly mutated.

Staying in the imperative family

In the procedural programming paradigm, data and behavior is independent from each other making access to data unaccounted and unpredictable.

Staying in the imperative family

Because of this every piece of data and every piece of behavior become mixed together.

There are no boundaries between irrelevant data.

Fundamental Concepts of OOP

Object oriented programming is usually defined using its three core design principles:

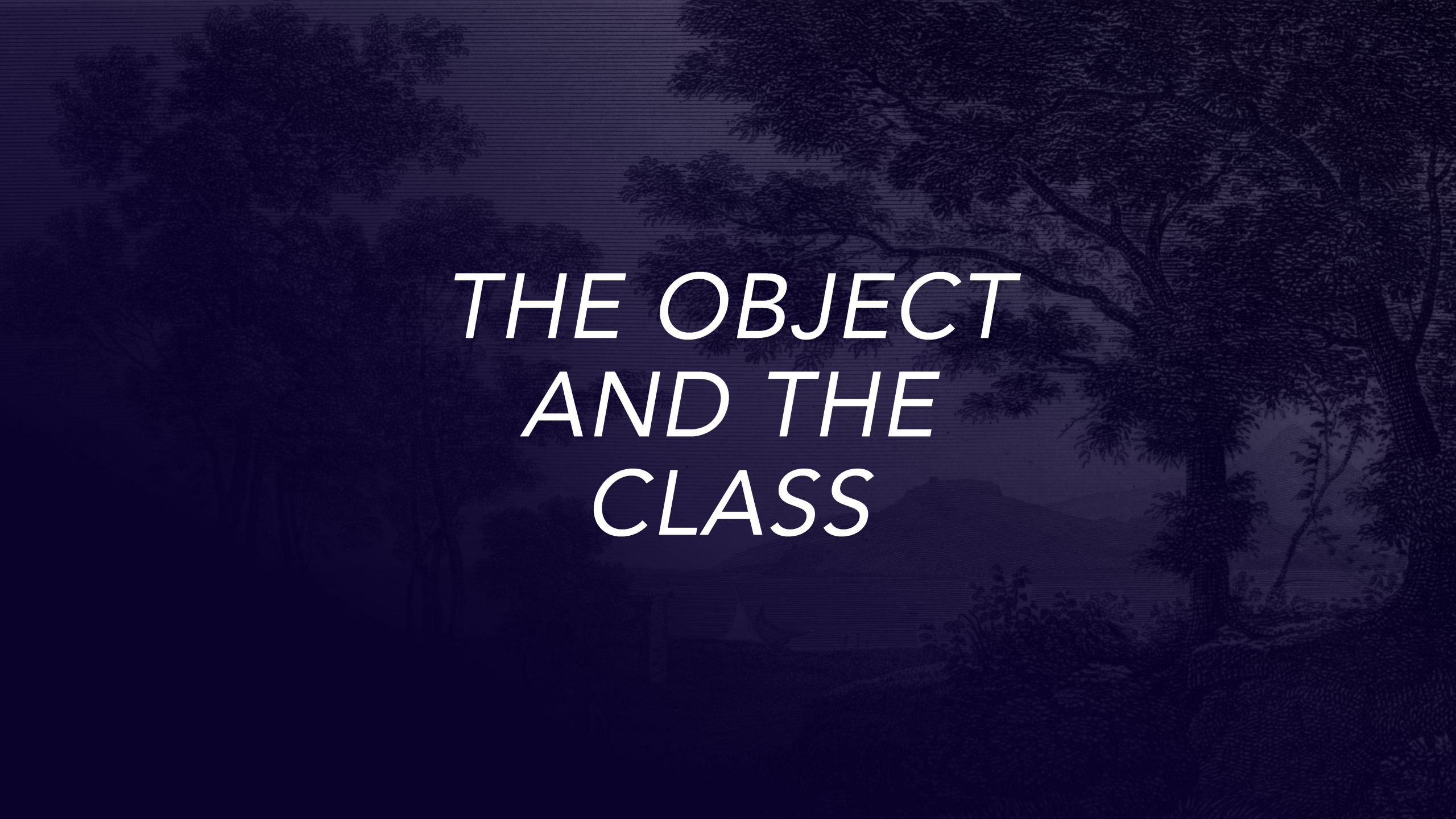
- Encapsulation
- Inheritance
- Polymorphism

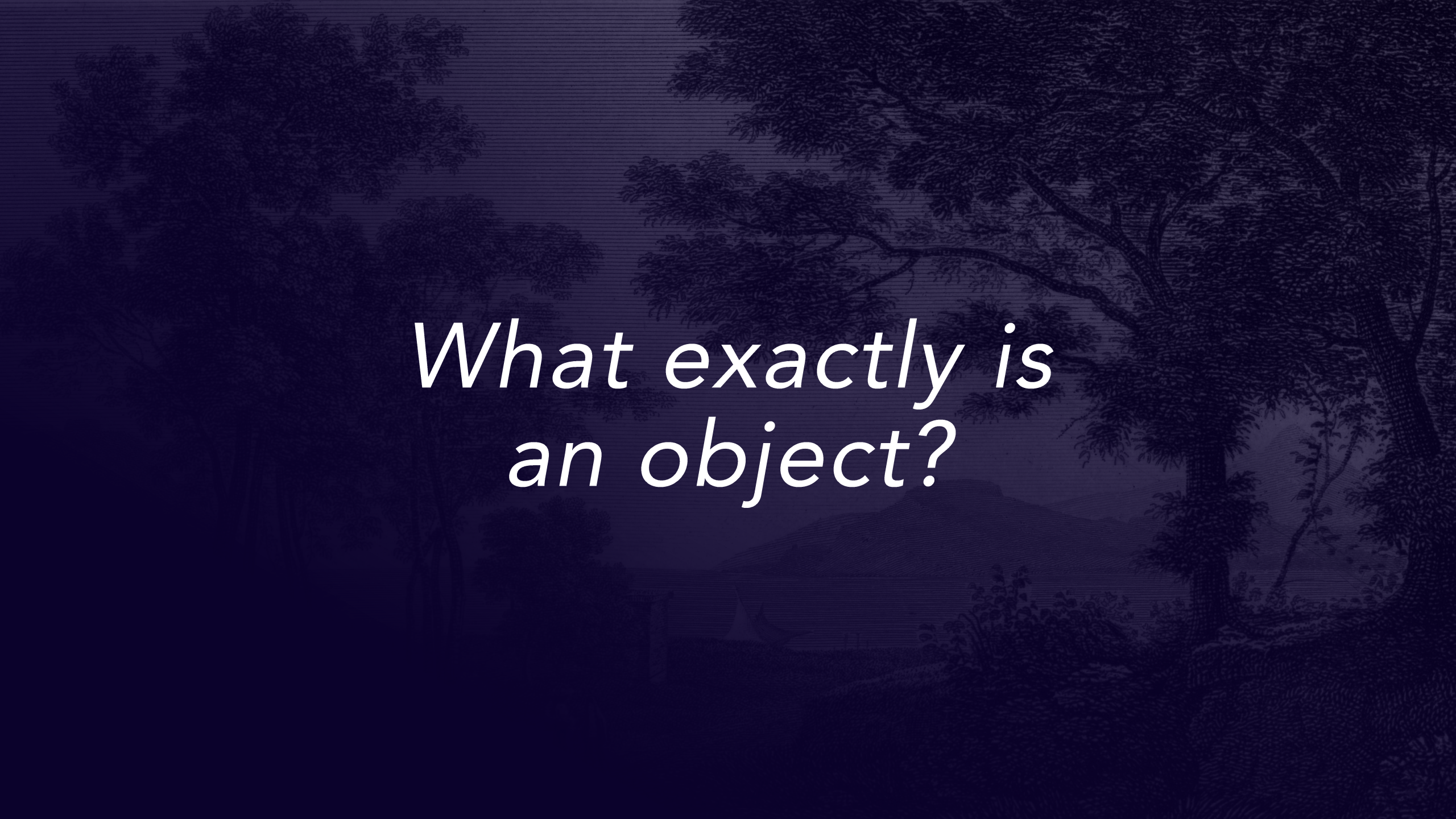
Fundamental Concepts

Calling the entirety of a programming language, OOP can be problematic since you are free to write code written in an "OOP language" without actually using these design principles.

Fundamental Concepts

At the end of the day OOP is not a language classification, but a paradigm.


THE OBJECT
AND THE
CLASS


What exactly is
an object?

Object

An object is a living organism in your code.

Treating an object as an organism will guide you on how you use objects effectively.

Just like any creature an object has both form (attributes) and behavior (methods).

Objects are written to be representations of real world nouns such as a person or an employee or a file.

Object

The best way to design objects is to simulate the real world form and behavior of what these objects represent.

Think of objects as the **representatives** of things in your code.

Class

There's usually a lot of confusion when identifying the difference between a class and an object and it is probably because in the universe of your code, the class and the object will have the same name.

A class is the specifications for the creation of objects.

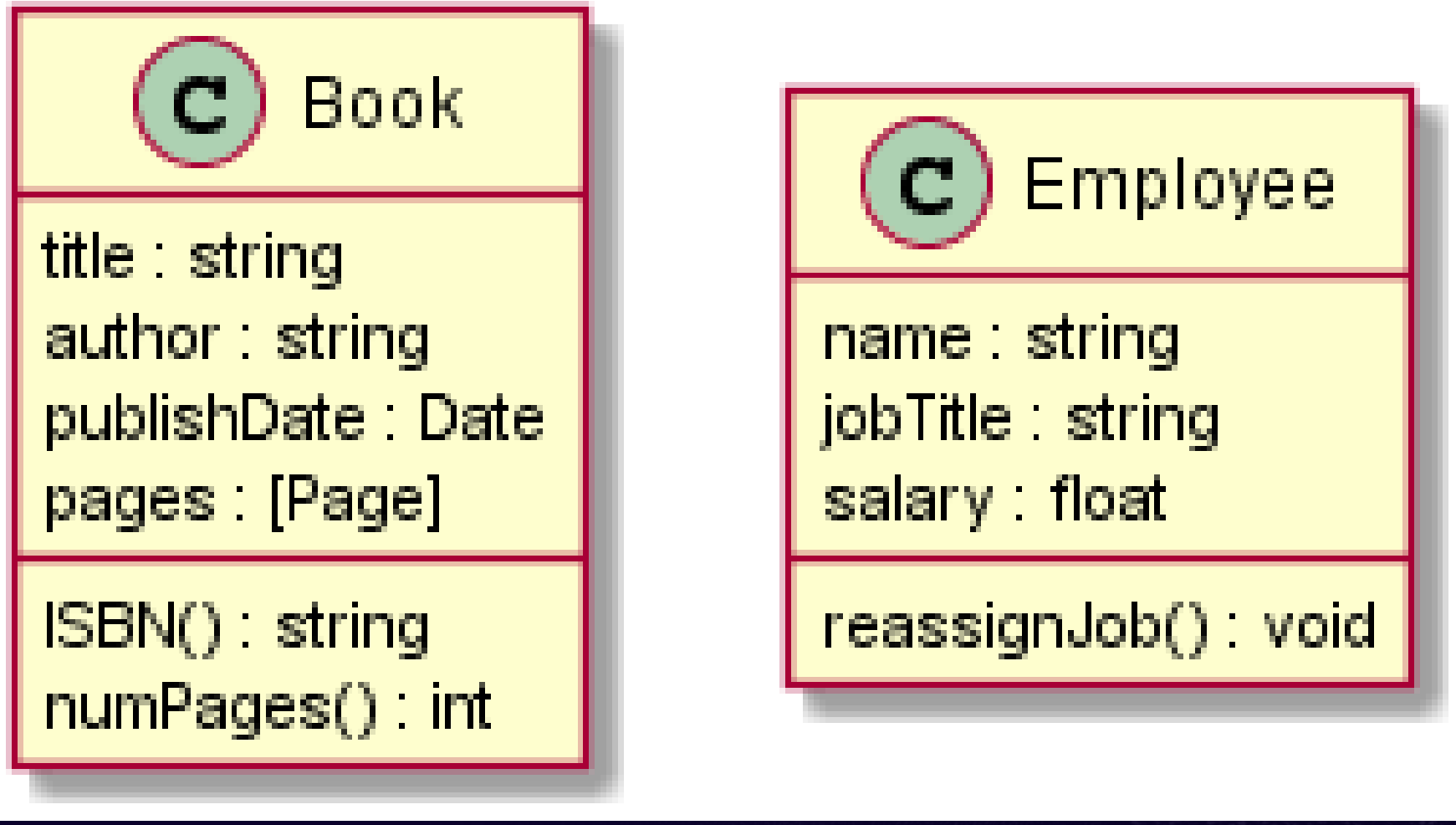
If you want to give a class a more proactive role, you can think of the class as the factory that builds objects.

Class

a star shaped cookie cutter (class) that makes star shaped cookies (objects). If you want to make star shaped cookies you use the star shaped cookie cutters.

Class

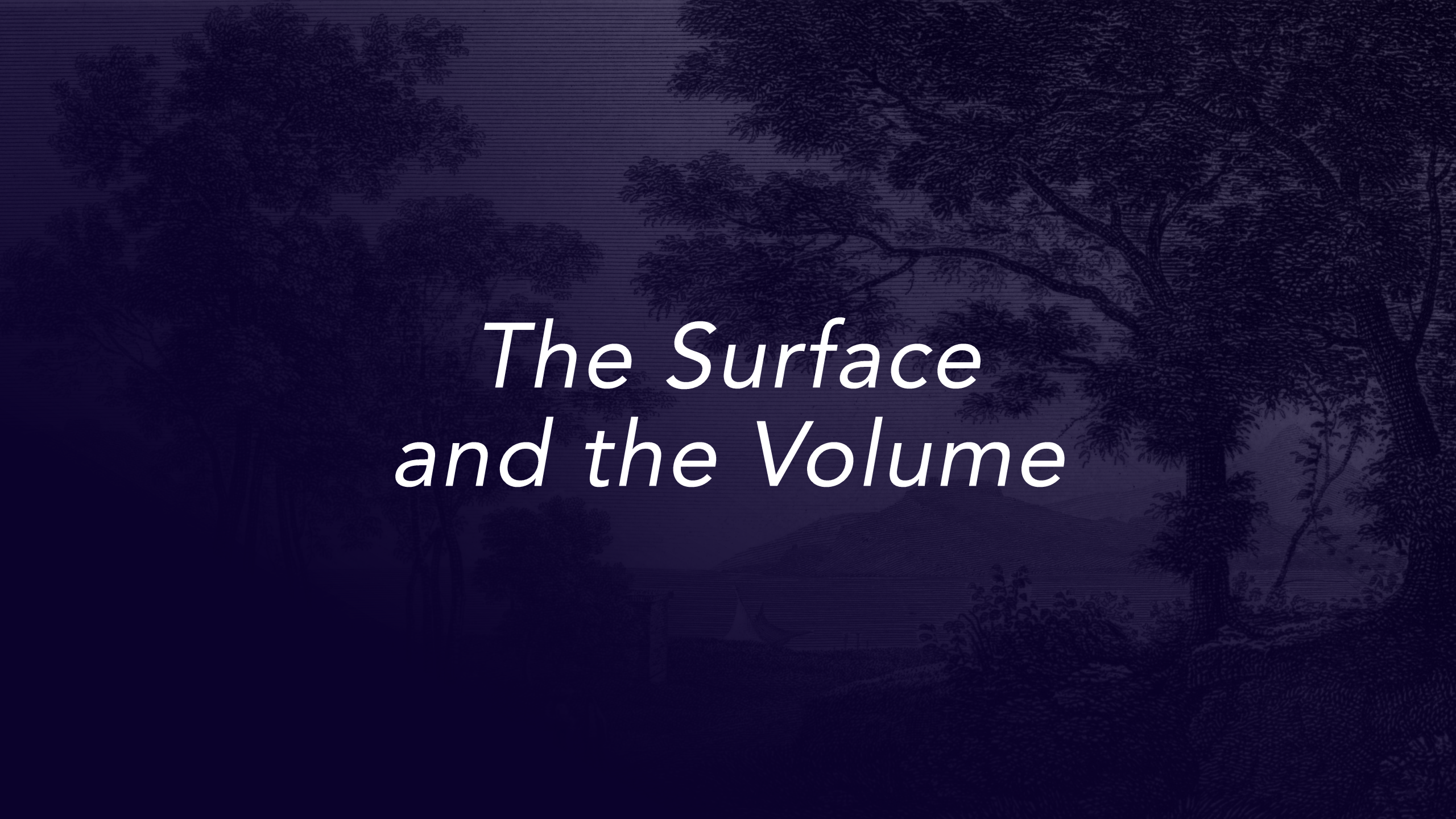
If an object is a representation of a tangible real world object, then the class is the conceptual type/category of that real world object.


Book
title : string
author : string
publishDate : Date
pages : [Page]
ISBN() : string
numPages() : int
Employee
name : string
jobTitle : string
salary : float
reassignJob() : void

book1{
title: "Corpus Hermeticum"
author: "Hermes Trismegistus“
publishDate: Dec 12, 2008
pages: ……
}

book2{
title: "Behold a Pale Horse"
author: "Milton William Cooper"
publishDate: Dec 1, 1991
pages: ……
}

employee1{
name: "Rubelito Abella"
jobTitle: "Instructor 1"
salary: [REDACTED]
}


The Surface
and the Volume

Data Hiding and the Interface

The base premise of OOP is the concept called **data hiding**.

Data Hiding and the Interface

What OOP did for imperative programming was to allow the programmer to create artificial boundaries between irrelevant data and behavior.

Data Hiding and the Interface

In the eyes of an OOP design, procedural code is a mix of data and functions arbitrarily tossed in a spaghetti of mutations and side-effects.

Data Hiding and the Interface

OOP's mechanism to create boundaries in the form of objects brought structure to imperative programming.

Data Hiding and the Interface

Object oriented programming obsesses over structure and simulation because it is a necessity for human comprehension and therefore, maintainability.

Data Hiding and the Interface

Systems need good structure not because well structured code is pleasant and elegant to look at, but because our feeble minds can't process poor structure efficiently.

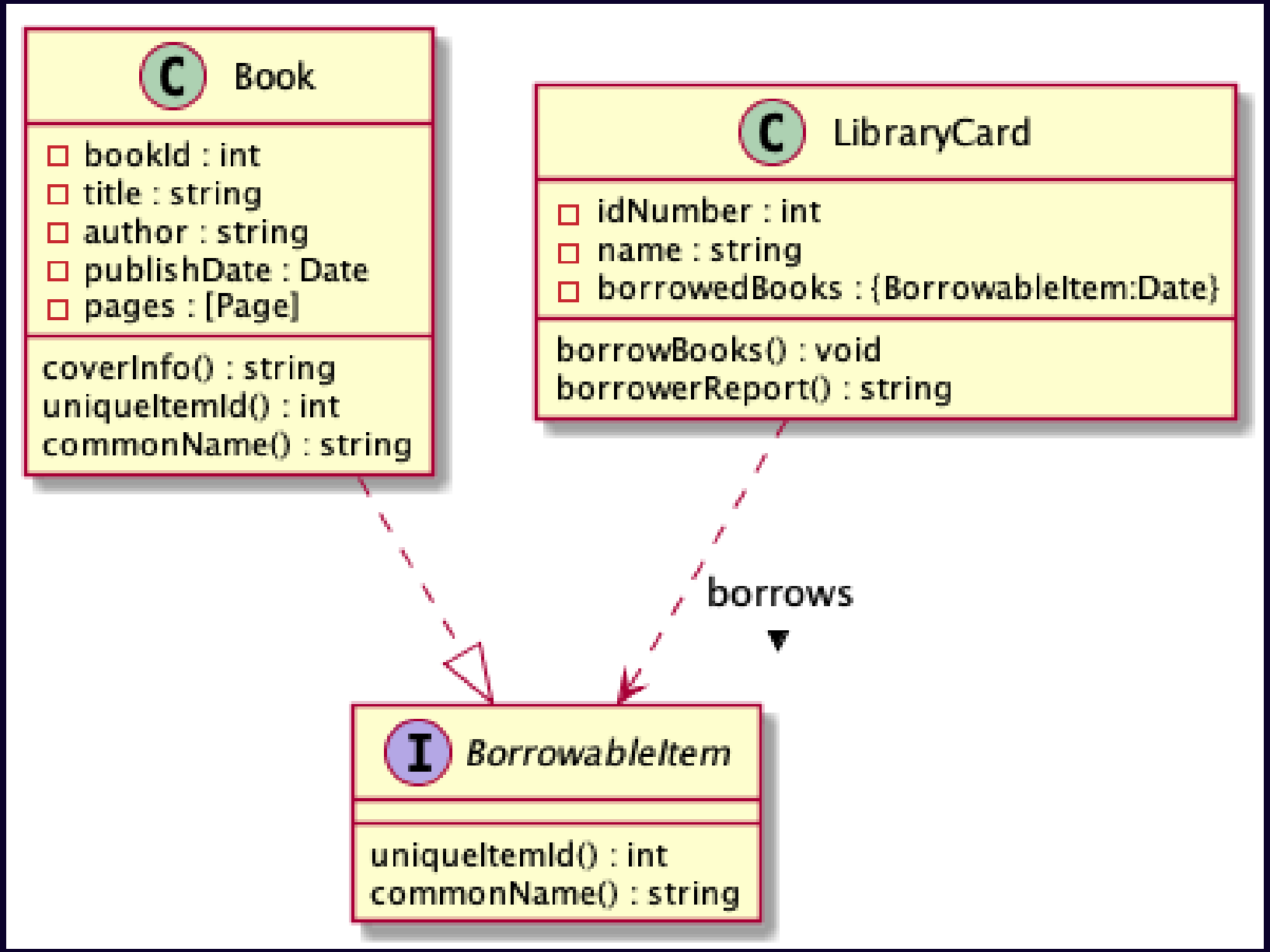
Data Hiding and the Interface

The process of modeling elegant object representations is basically determining what the **public interface** of that object may be.

The interface of an object is the set of attributes and methods that other objects can use to interact with it.

Data Hiding and the Interface

The attributes and methods that are not in the interface are essentially hidden information, inaccessible from the client objects.


Book
bookId : int
title : string
author : string
publishDate : Date
pages : [Page]
coverInfo() : string
uniqueItemId() : int
commonName() : string
LibraryCard
idNumber : int
name : string
borrowedBooks : {BorrowableItem:Date}
borrowBooks() : void
borrowerReport() : string
borrows
BorrowableItem
uniqueItemId() : int
commonName() : string

Abstraction of Objects

Creating interfaces like these provide OOP with the mechanism to create **abstractions** in the object level.

Abstraction of Objects

An abstraction in computer science is basically a model of computation that is free from its implementation.

Abstraction of Objects

In the same way that functional programming creates abstractions of mathematical functions by writing lambdas without side effects, OOP creates abstractions of objects using interfaces that don't specify the exact implementation of an object.

Abstraction of Objects

The reason why this structure still works, is because we have a concrete class called `Book` which is a **realization** or an **implementation** of `BorrowableItem`.

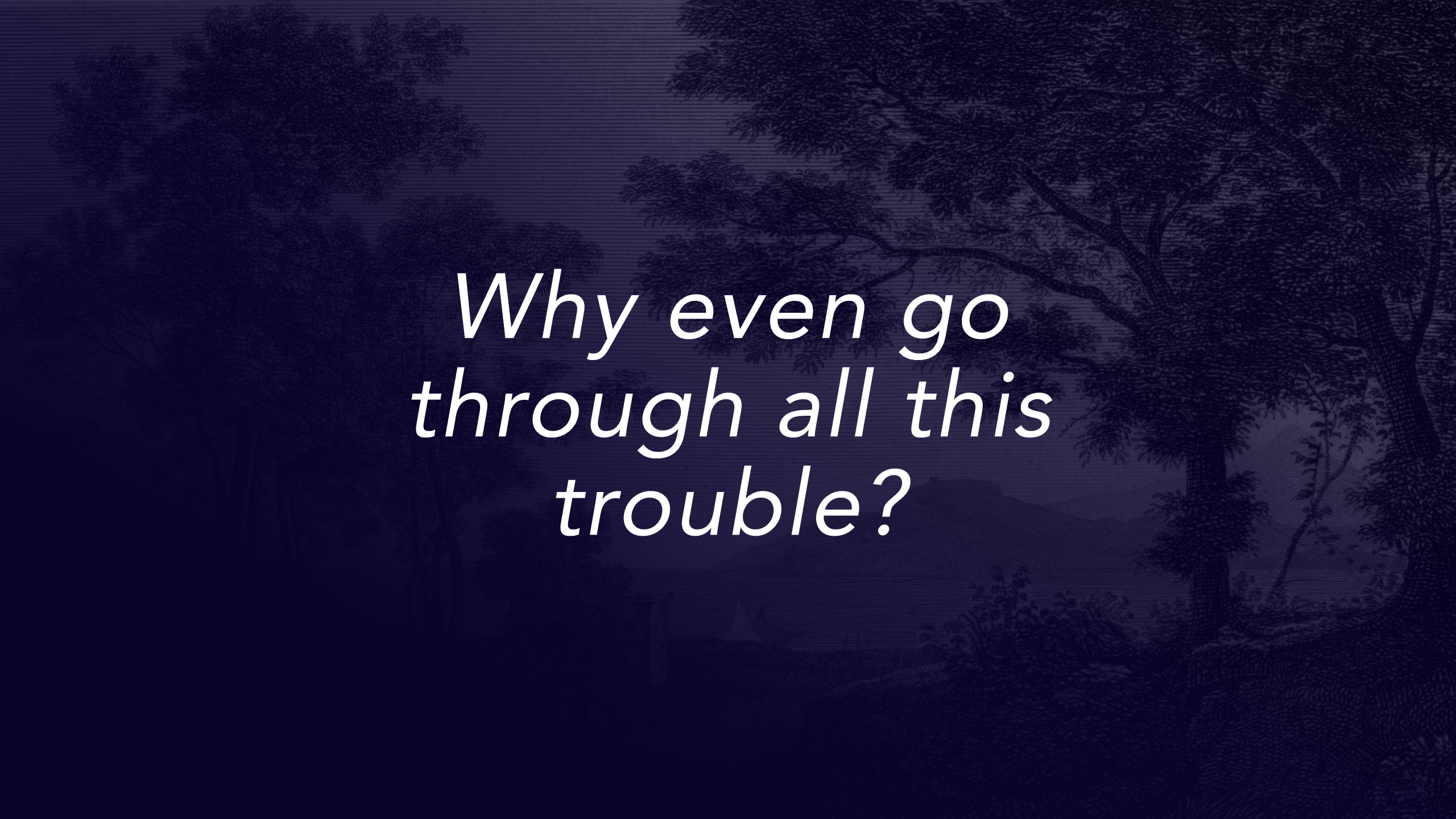
Abstraction of Objects

Because a Book is a BorrowableItem, it must also behave based on the specifications of a BorrowableItem.

Meaning it must contain the methods uniqueItemId() and commonName()

Abstraction of Objects

Since Book is a concrete class it's methods uniqueItemId() and commonName() should be implemented (meaning there should be code inside these methods).


Why even go
through all this
trouble?

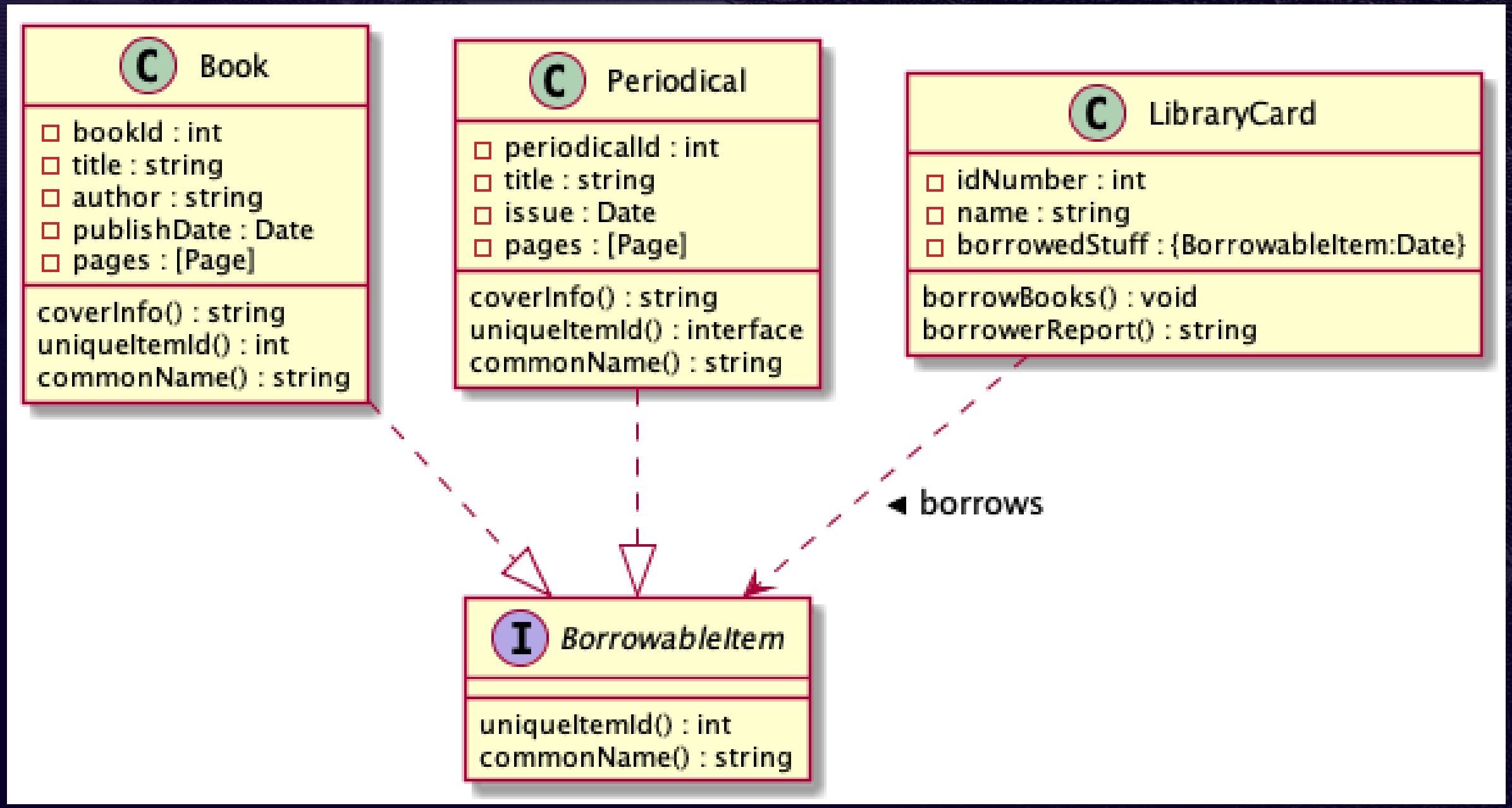
Abstraction and maintainability

For the current structure we created, this feels like extra code because our system is small enough right now.

Abstraction and maintainability

For the current structure we created, this feels like extra code because our system is small enough right now.

If our system grows and we need to incorporate other things from the library that are not books but can be borrowed.


Book
bookId : int
title : string
author : string
publishDate : Date
pages : [Page]
coverInfo() : string
uniqueItemId() : int
commonName() : string
Periodical
periodicalId : int
title : string
issue : Date
pages : [Page]
coverInfo() : string
uniqueItemId() : interface
commonName() : string
LibraryCard
idNumber : int
name : string
borrowedStuff : {BorrowableItem:Date}
borrowBooks() : void
borrowerReport() : string
borrows
BorrowableItem
uniqueItemId() : int
commonName() : string

Abstraction and maintainability

You need a different representation for a periodical, therefore you need to create a new concrete class called `Periodical`.

Abstraction and maintainability

Since a periodical is also **something from the library that can be borrowed**, a periodical is another **realization** of `BorrowableItem`.

Abstraction and maintainability

And with the tiny effort of writing the implementation of a periodical (including the realized methods `uniqueItemId()` and `commonName()`), we added an extra interaction that allows a `LibraryCard` to borrow periodicals as well.

OOP's Philosophy The Interface and the Implementation

The paradigm's aims to solve the issues of state and maintainability by allowing programmers to create boundaries between its mix of attributes and methods.

OOP's Philosophy: The Interface and the Implementation

The boundaries you enforce are basically the object structure you create.

A library card name shouldn't mix with a book title so we put a boundary between them by **encapsulating** them into their respective objects.

OOP's Philosophy: The Interface and the Implementation

The boundaries you enforce are basically the object structure you create.

A library card name shouldn't mix with a book title so we put a boundary between them by **encapsulating** them into their respective objects.

OOP's Philosophy: The Interface and the Implementation

The boundaries you enforce are basically the object structure you create.

A library card name shouldn't mix with a book title so we put a boundary between them by **encapsulating** them into their respective objects.