

OOP IN DART

IT'S ALL ABOUT OBJECTS

FEW FACTS ABOUT CLASSES IN DART

FEW FACTS ABOUT CLASSES IN DART

- ▶ Every object in Dart is an instance of a class

FEW FACTS ABOUT CLASSES IN DART

- ▶ Every object in Dart is an instance of a class
- ▶ Every class inherit from Object

FEW FACTS ABOUT CLASSES IN DART

- ▶ Every object in Dart is an instance of a class
- ▶ Every class inherit from Object
- ▶ Each class can have only one superclass

FEW FACTS ABOUT CLASSES IN DART

- ▶ Every object in Dart is an instance of a class
- ▶ Every class inherit from Object
- ▶ Each class can have only one superclass
- ▶ Classes can have properties and methods

FEW FACTS ABOUT CLASSES IN DART

- ▶ Every object in Dart is an instance of a class
- ▶ Every class inherit from Object
- ▶ Each class can have only one superclass
- ▶ Classes can have properties and methods
- ▶ Both methods and properties can be instance or class (static)

WE KNOW YOU

```
class Movie {  
    final String name;  
  
    Movie({required this.name});  
  
    void play() {  
        print('Playing movie $name');  
    }  
}
```


WE KNOW YOU

```
class Movie {  
    final String name;  
  
    Movie({required this.name});  
  
    void play() {  
        print('Playing movie $name');  
    }  
}
```

```
var movie = Movie(name: 'Thor: Love and Thunder');  
movie.play();
```

PROPERTIES

```
class Movie {  
  final String name; // must have a value when created  
  int? productionYear; // initial value is null  
  List<String> castMembers = []; // initial empty value  
  late double rating; // same as final, but can be created later  
}
```

PROPERTIES

```
class Movie {  
  final String name; // must have a value when created  
  int? productionYear; // initial value is null  
  List<String> castMembers = []; // initial empty value  
  late double rating; // same as final, but can be created later
```

```
static const maximumDuration = 300;|
```

PROPERTIES

```
class Movie {  
  final String name; // must have a value when created  
  int? productionYear; // initial value is null  
  List<String> castMembers = []; // initial empty value  
  late double rating; // same as final, but can be created later
```

```
static const maximumDuration = 300;
```

```
int get yearsInProduction => 2022 - productionYear!;
```

CONSTRUCTORS

```
Movie(String name) {  
  this.name = name;  
}
```

CONSTRUCTORS

```
Movie(String name) {  
  this.name = name;  
}
```

```
Movie(this.name);
```

CONSTRUCTORS

```
Movie(String name) {  
  this.name = name;  
}
```

```
var movie = Movie('Iron Man');
```

```
Movie(this.name);
```

CONSTRUCTORS

```
Movie(String name) {  
  this.name = name;  
}
```

```
var movie = Movie('Iron Man');
```

```
Movie(this.name);
```

```
Movie({required this.name});
```

```
Movie(this.productionYear, {required this.name});
```


CONSTRUCTORS

```
Movie(String name) {  
  this.name = name;  
}
```

```
var movie = Movie('Iron Man');
```

```
Movie(this.name);
```

```
Movie({required this.name});
```

```
var movie = Movie(name: 'Ant Man');
```

```
Movie(this.productionYear, {required this.name});
```

```
var movie2 = Movie(2016, name: 'Strange');
```

CONSTRUCTORS

```
var movie = Movie(); // default constructor
```

CONSTRUCTORS

```
var movie = Movie(); // default constructor
```

```
// named constructor  
Movie.withProductionYear(String name, int productionYear)  
    : name = name,  
      productionYear = productionYear;  
|
```

```
var movie = Movie.withProductionYear('Doctor Strange', 2016);
```

CONSTRUCTORS

```
var movie = Movie(); // default constructor
```

```
// named constructor  
Movie.withProductionYear(String name, int productionYear)  
    : name = name,  
      productionYear = productionYear;  
|
```

```
var movie = Movie.withProductionYear('Doctor Strange', 2016);
```

Constructors are not inherited

CONSTRUCTORS

You can validate parameters

```
// named constructor  
Movie.withProductionYear(String name, int productionYear)  
  : assert(productionYear > 2008),  
    name = name,  
    productionYear = productionYear;
```

CONSTRUCTORS

You can validate parameters

```
// named constructor  
Movie.withProductionYear(String name, int productionYear)  
  : assert(productionYear > 2008),  
    name = name,  
    productionYear = productionYear;
```

Debug only

CONSTRUCTORS

You can validate parameters

```
// named constructor  
Movie.withProductionYear(String name, int productionYear)  
    : assert(productionYear > 2008),  
      name = name,  
      productionYear = productionYear;
```

Debug only

```
Movie.redirected(String name) : this.withProductionYear(name, 2020);
```

CONSTRUCTORS SUMMARY

```
Movie(String name) {  
  this.name = name;  
}
```

```
var movie = Movie(); // default constructor
```

```
Movie(this.name);
```

```
// named constructor  
Movie.withProductionYear(String name, int productionYear)  
  : name = name,  
    productionYear = productionYear;  
|
```

```
Movie({required this.name});
```

```
Movie(this.productionYear, {required this.name});
```

```
Movie.redirected(String name) : this.withProductionYear(name, 2020);
```

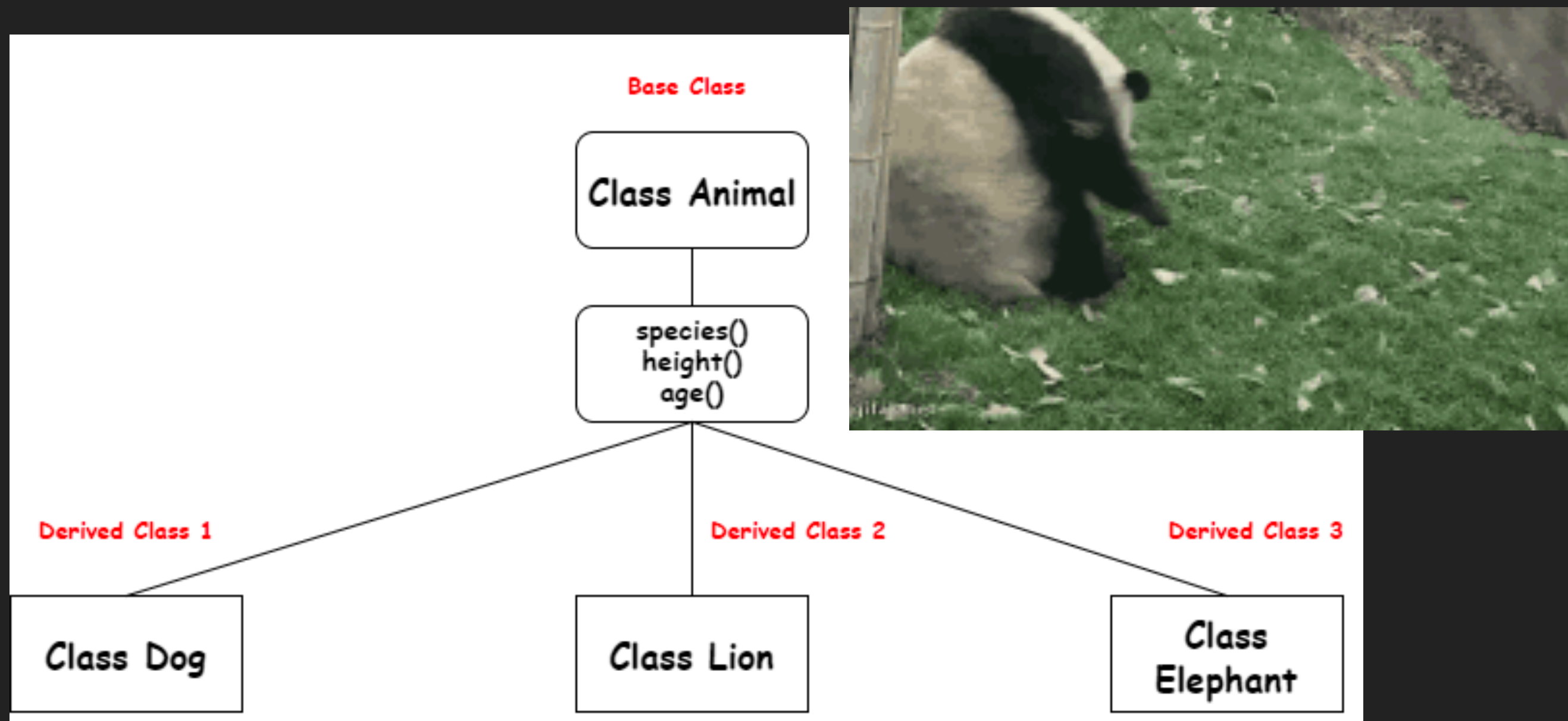

OOP

OOP

HOW DO WE LEARN IT?

FIRST WE LEARN ABOUT ANIMALS

FIRST WE LEARN ABOUT ANIMALS



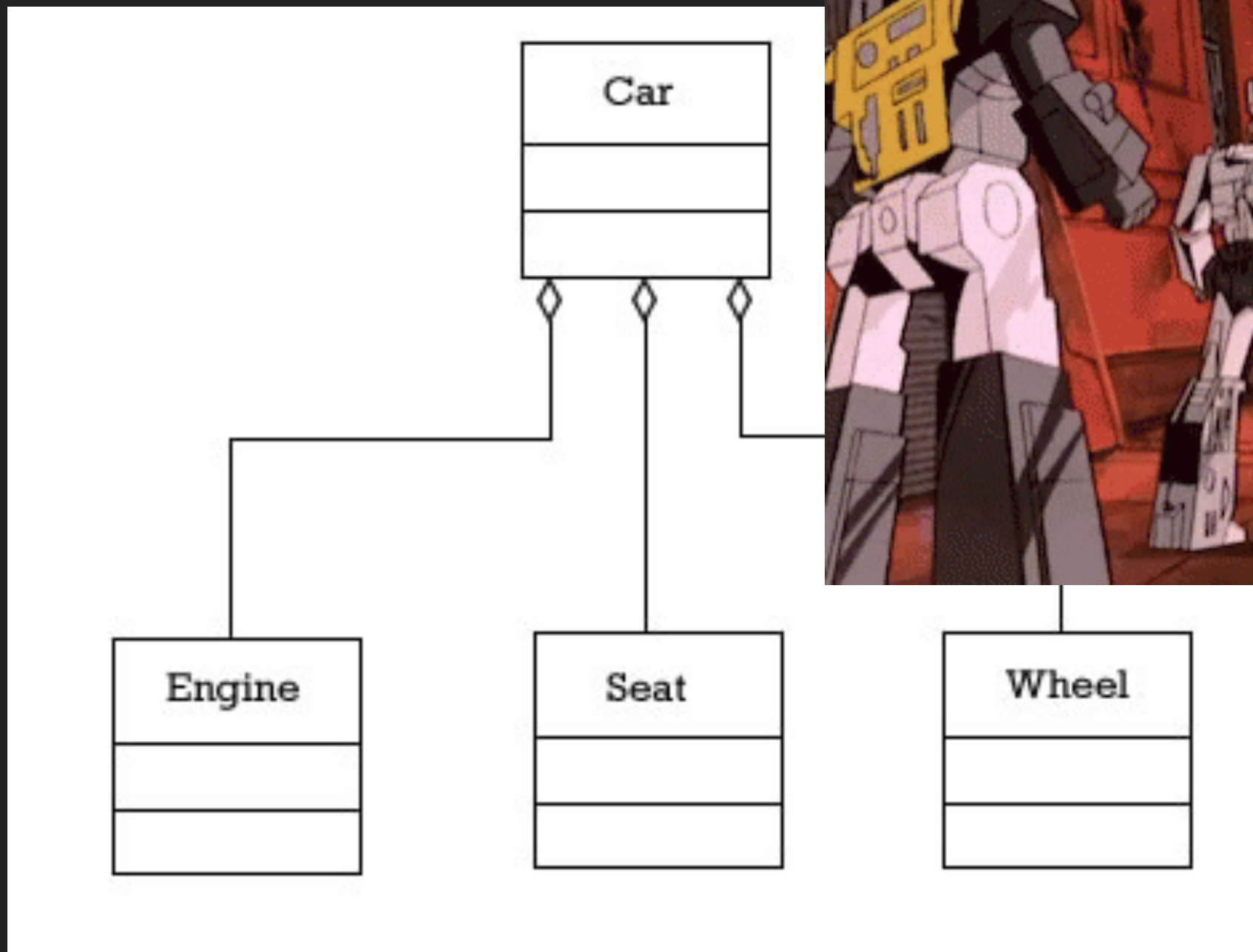
THEN ABOUT CARS

THEN ABOUT CARS



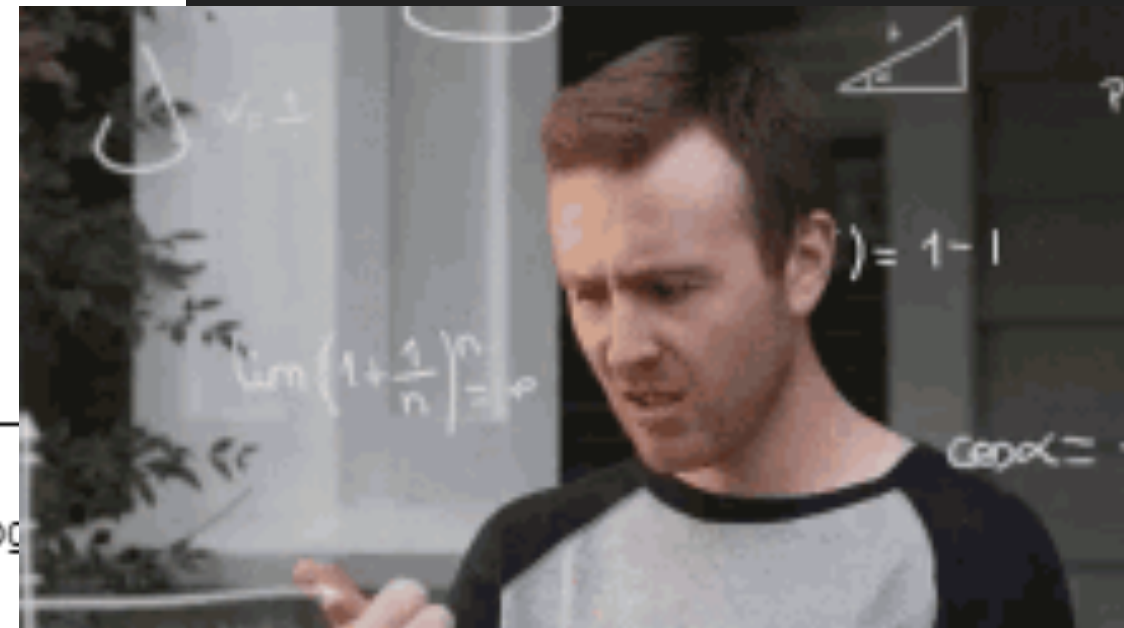
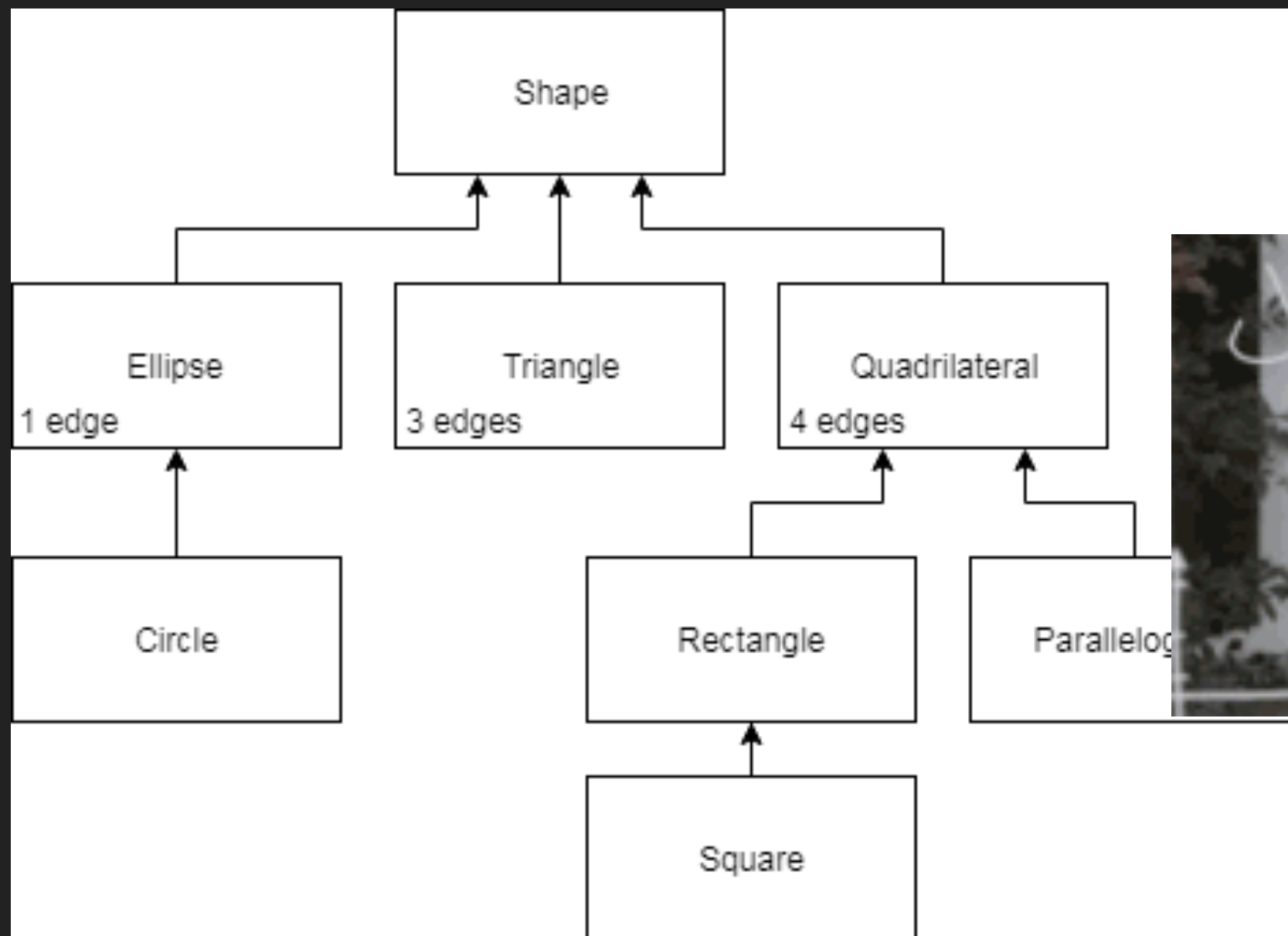
AND COMPOSITION

AND COMPOSITION



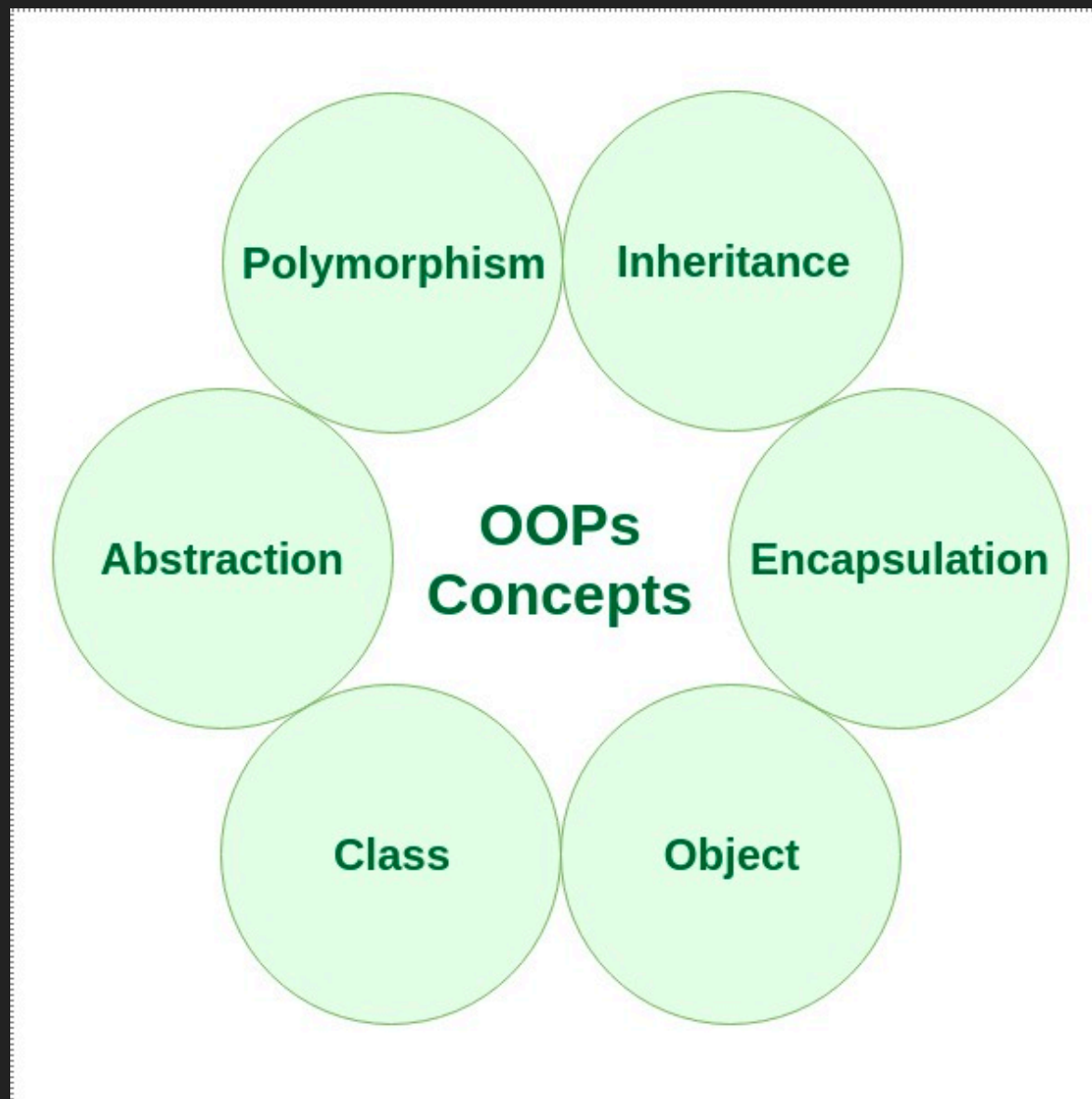
AND THEN FIGURES

AND THEN FIGURES



AND FINALLY WE HAVE THIS

AND FINALLY WE HAVE THIS



MANY THEORY

**MANY THEORY
FAR FROM REAL
PROGRAMMING**

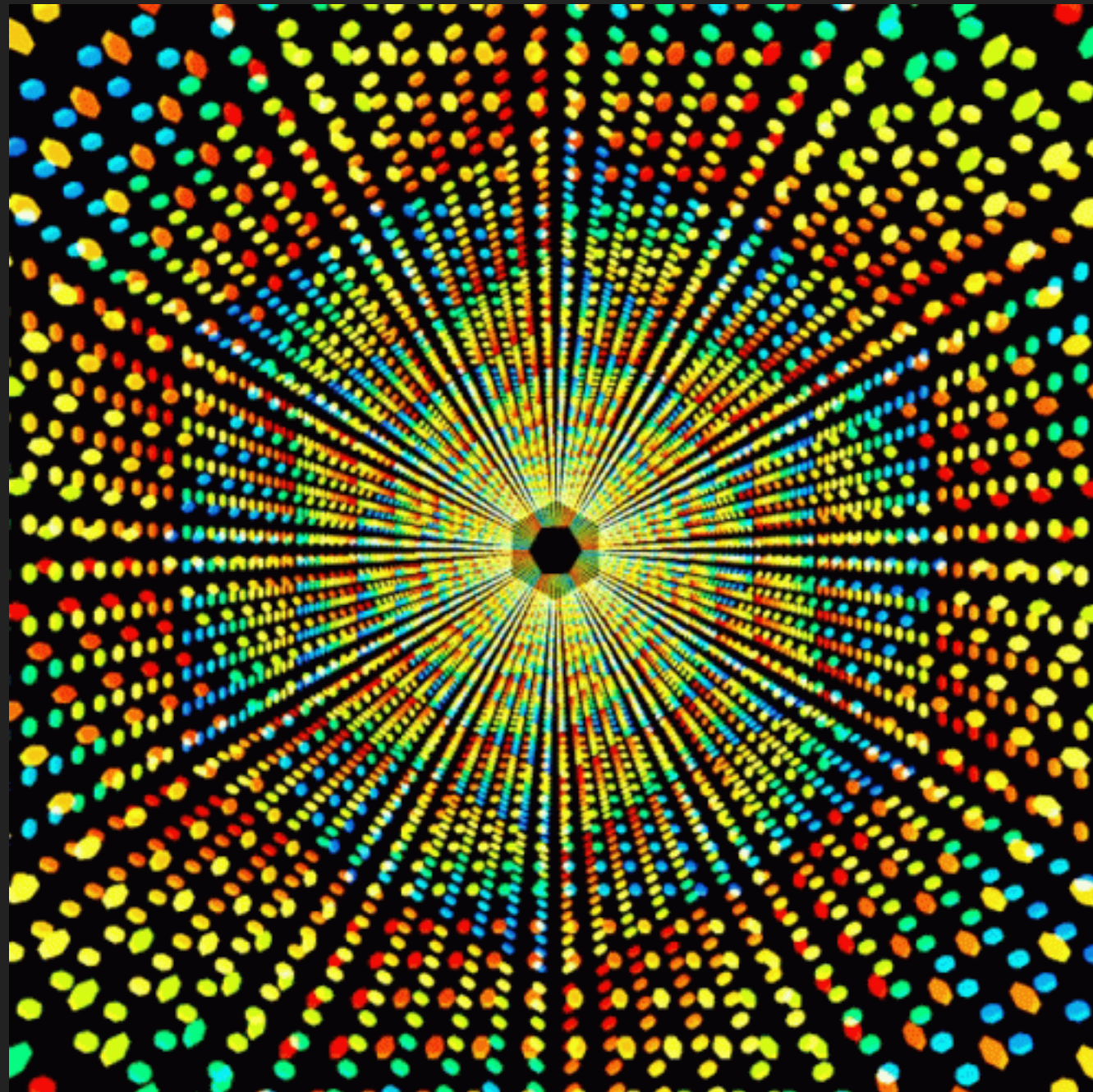


LET'S DO SOME REAL SH*T

LET'S DO SOME REAL SH*T



0. ABSTRACTION



0. ABSTRACTION

- ▶ Our code is separated into different pieces that are doing it's own work

0. ABSTRACTION

- ▶ Our code is separated into different pieces that are doing it's own work
- ▶ If it was all together - it would be a total mess

0. ABSTRACTION

- ▶ Our code is separated into different pieces that are doing it's own work
- ▶ If it was all together - it would be a total mess
- ▶ It's more about general concept of organising code and how your program works

0. ABSTRACTION

- ▶ Our code is separated into different pieces that are doing it's own work
- ▶ If it was all together - it would be a total mess
- ▶ It's more about general concept of organising code and how your program works
- ▶ Making separate widgets, moving logic code to separate classes is also doing abstractions

1. ENCAPSULATION



1. ENCAPSULATION

- ▶ We don't know about implementation details

1. ENCAPSULATION

- ▶ We don't know about implementation details
- ▶ We have limited control

1. ENCAPSULATION

- ▶ We don't know about implementation details
- ▶ We have limited control
- ▶ Let's us focus on final result

1. ENCAPSULATION

- ▶ We don't know about implementation details
- ▶ We have limited control
- ▶ Let's us focus on final result
- ▶ Internal logic is responsibility of feature provider (service)

1. ENCAPSULATION

- ▶ We don't know about implementation details
- ▶ We have limited control
- ▶ Let's us focus on final result
- ▶ Internal logic is responsibility of feature provider (service)
- ▶ Most importantly we don't care about internal logic and it's complexity - we just believe that result will be good for us

1. ENCAPSULATION

- ▶ We don't know about implementation details
- ▶ We have limited control
- ▶ Let's us focus on final result
- ▶ Internal logic is responsibility of feature provider (service)
- ▶ Most importantly we don't care about internal logic and it's complexity - we just believe that result will be good for us
- ▶ Less possible interactions -> less bugs in code

1. ENCAPSULATION

- ▶ We ordered burger in restaurant - we don't care how chef is gonna make it (give example code here)

1. ENCAPSULATION

- ▶ We ordered burger in restaurant - we don't care how chef is gonna make it (give example code here)
- ▶ We order Glovo from restaurant - we don't care about all internal logic - how it finds a courier, how it communicates with restaurant - we just want to eat

1. ENCAPSULATION

- ▶ We ordered burger in restaurant - we don't care how chef is gonna make it (give example code here)
- ▶ We order Glovo from restaurant - we don't care about all internal logic - how it finds a courier, how it communicates with restaurant - we just want to eat
- ▶ We come to dry cleaner - we don't care about what detergent will be used, how long it will be washed - we just want clear stuff

1. ENCAPSULATION

```
class Restaurant {  
  List<Table> tables = [  
    Table(numberOfSeats: 4, state: TableState.free),  
    Table(numberOfSeats: 5, state: TableState.free),  
    Table(numberOfSeats: 6, state: TableState.occupied),  
    Table(numberOfSeats: 4, state: TableState.occupied),  
  ];  
}
```

1. ENCAPSULATION

```
class Restaurant {  
  List<Table> tables = [  
    Table(numberOfSeats: 4, state: TableState.free),  
    Table(numberOfSeats: 5, state: TableState.free),  
    Table(numberOfSeats: 6, state: TableState.occupied),  
    Table(numberOfSeats: 4, state: TableState.occupied),  
  ];  
}
```

```
class Table {  
  int numberOfSeats;  
  TableState state;  
  
  Table({required this.numberOfSeats, required this.state});  
}
```

```
enum TableState {  
  free, reserved, occupied  
}
```

1. ENCAPSULATION

```
var restaurant = Restaurant();  
restaurant.tables[0].state = TableState.occupied;
```

1. ENCAPSULATION

```
var restaurant = Restaurant();  
restaurant.tables[0].state = TableState.occupied;
```

Very rude client

1. ENCAPSULATION

```
var restaurant = Restaurant();  
restaurant.tables[0].state = TableState.occupied;
```

Very rude client

```
var freeTable = restaurant.tables.firstWhere((table) {  
    return (table.numberOfSeats >= 10 && table.state == TableState.free );  
});  
  
if (freeTable != null) {  
    print('We have table! Yaya');  
} else {  
    print('No tables for us');  
}
```

1. ENCAPSULATION

```
var restaurant = Restaurant();  
restaurant.tables[0].state = TableState.occupied;
```

Very rude client

```
var freeTable = restaurant.tables.firstWhere((table) {  
    return (table.numberOfSeats >= 10 && table.state == TableState.free );  
});  
  
if (freeTable != null) {  
    print('We have table! Yaya');  
} else {  
    print('No tables for us');  
}
```

More polite client

1. ENCAPSULATION

Let's fix it!

1. ENCAPSULATION

Let's fix it!

```
List<Table> _tables =
```


1. ENCAPSULATION

Let's fix it!

```
List<Table> _tables =
```

```
var restaurant = Restaurant();  
restaurant.tables[0].state = TableState.occupied;
```

The getter 'tables' isn't defined for the type 'Restaurant'.

1. ENCAPSULATION

Let's fix it!

```
List<Table> _tables =
```

```
var restaurant = Restaurant();  
restaurant.tables[0].state = TableState.occupied;
```

The getter 'tables' isn't defined for the type 'Restaurant'.

Oh snap...

1. ENCAPSULATION

```
Table? availableTable({required int visitorsCount}) {  
  var freeTable = _tables.firstWhere((table) {  
    return (table.numberOfSeats >= visitorsCount && table.state == TableState.free);  
  });  
  
  if (freeTable != null) {  
    return freeTable;  
  } else {  
    return null;  
  }  
}
```

1. ENCAPSULATION

```
Table? availableTable({required int visitorsCount}) {  
  var freeTable = _tables.firstWhere((table) {  
    return (table.numberOfSeats >= visitorsCount && table.state == TableState.free);  
  });  
  
  if (freeTable != null) {  
    return freeTable;  
  } else {  
    return null;  
  }  
}
```

```
void occupyTable(Table table) {  
  var index = _tables.indexOf(table);  
  _tables[index].state = TableState.occupied;  
}
```

1. ENCAPSULATION

```
var restaurant = Restaurant();  
var freeTable = restaurant.availableTable(visitorsCount: 4);  
if (freeTable != null) {  
    restaurant.occupyTable(freeTable);  
}
```

1. ENCAPSULATION

```
var restaurant = Restaurant();  
var freeTable = restaurant.availableTable(visitorsCount: 4);  
if (freeTable != null) {  
    restaurant.occupyTable(freeTable);  
}
```

Now we have limited access
and that makes our system
more safe and predictable

1. ENCAPSULATION

We can also apply it to our widgets - make them private and don't let other modify them or call functions on them. And the most important part is how we build model layer with this in mind

2. INHERITANCE

2. INHERITANCE

- ▶ Use **extends** to create a subclass

2. INHERITANCE

- ▶ Use **extends** to create a subclass
- ▶ Use **super** to refer to superclass

2. INHERITANCE

- ▶ Use **extends** to create a subclass
- ▶ Use **super** to refer to superclass

```
class ParentClass {  
    void _prepareMainEngine() {  
        print('do something important');  
    }  
  
    void turnOn() {  
        _prepareMainEngine();  
    }  
}
```

2. INHERITANCE

- ▶ Use **extends** to create a subclass
- ▶ Use **super** to refer to superclass

```
class ParentClass {  
  void _prepareMainEngine() {  
    print('do something important');  
  }  
  
  void turnOn() {  
    _prepareMainEngine();  
  }  
}
```

```
class Subclass extends ParentClass {  
  void _prepareAdditionalEngines() {  
    print('do something important on top of super');  
  }  
  
  void turnOn() {  
    super.turnOn();  
    _prepareAdditionalEngines();  
  }  
}
```

2. INHERITANCE

- ▶ Super class is nice for writing some base logic for subclasses

2. INHERITANCE

- ▶ Super class is nice for writing some base logic for subclasses
- ▶ Stateless and Stateful widgets are nice examples of base classes

2. INHERITANCE

- ▶ Super class is nice for writing some base logic for subclasses
- ▶ Stateless and Stateful widgets are nice examples of base classes
- ▶ You can override methods

2. INHERITANCE

- ▶ Super class is nice for writing some base logic for subclasses
- ▶ Stateless and Stateful widgets are nice examples of base classes
- ▶ You can override methods
- ▶ You can't inherit more than 1 superclass

2. INHERITANCE

- ▶ Super class is nice for writing some base logic for subclasses
- ▶ Stateless and Stateful widgets are nice examples of base classes
- ▶ You can override methods
- ▶ You can't inherit more than 1 superclass
- ▶ It's not better than composition :)

3. POLYMORPHISM

3. POLYMORPHISM

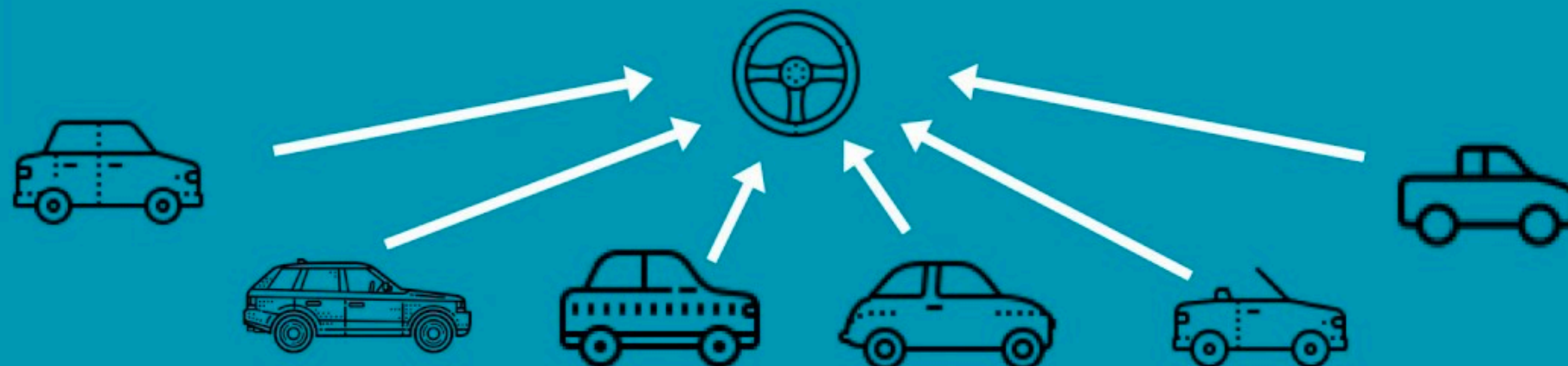
Polymorphism

one interface - multiple implementations

Polymorphism in Object-Oriented Programming is the ability to create a property, a function, or an object that has more than one realization.

Polymorphism is an ability to substitute classes that have common functionality in sense of methods and data.

If you learned driving one car, you'll be able to drive on any car, it doesn't depend on car brand or inner implementation. It has the same driver interface.



3. POLYMORPHISM

- ▶ We don't focus on implementation, we focus on ability

3. POLYMORPHISM

- ▶ We don't focus on implementation, we focus on ability
- ▶ It builds abstraction layer

3. POLYMORPHISM

- ▶ We don't focus on implementation, we focus on ability
- ▶ It builds abstraction layer
- ▶ When overriding methods we can take the best from base class and add our own behaviour

3. POLYMORPHISM

- ▶ We don't focus on implementation, we focus on ability
- ▶ It builds abstraction layer
- ▶ When overriding methods we can take the best from base class and add our own behaviour
- ▶ We can group different classes and make one action with them all -> Each class handles it individually

3. POLYMORPHISM

- ▶ We don't focus on implementation, we focus on ability
- ▶ It builds abstraction layer
- ▶ When overriding methods we can take the best from base class and add our own behaviour
- ▶ We can group different classes and make one action with them all -> Each class handles it individually
- ▶ Perfect example is how we add widgets to row or column

3. POLYMORPHISM

```
// You can't create me :)
abstract class Animal {
    void move();
}

class Fish implements Animal {
    void move() {
        print('I am swimming in the ocean yall');
    }
}

class Bird implements Animal {
    void move() {
        print('I can see the skyyyyyy');
    }
}
```

3. POLYMORPHISM

```
// You can't create me :)
abstract class Animal {
    void move();
}

class Fish implements Animal {
    void move() {
        print('I am swimming in the ocean yall');
    }
}

class Bird implements Animal {
    void move() {
        print('I can see the skyyyyyy');
    }
}
```

```
List<Animal> animals = [Fish(), Mammal(), Bird(), Mammal()];
animals.forEach((animal) => animal.move());
```

3. POLYMORPHISM

It is extremely helpful when
injecting different services that
can have different
implementations based on the
conditions

3. POLYMORPHISM

- ▶ Class can implement multiple interfaces

3. POLYMORPHISM

- ▶ Class can implement multiple interfaces
- ▶ Each class have own implicit interface

```
class A {  
    void greet() {}  
  
    void play() {}  
}  
  
class B implements A {  
    void greet() {}  
  
    void play() {}  
}
```

3. POLYMORPHISM

Using interfaces we focus on high level

```
abstract class DatabaseService {  
    void save(DatabaseObject object);  
    void update(DatabaseObject object);  
    void delete(DatabaseObject object);  
    List<DatabaseObject> getObjects();  
}
```

We care only about actions

MIXINS

MIXINS

A way to reuse code in multiple class hierarchies

MIXINS

A way to reuse code in multiple class hierarchies

```
class Musician extends Performer with Musical {  
    // ...  
}  
  
class Maestro extends Person with Musical, Aggressive, Demented {  
    Maestro(String maestroName) {  
        name = maestroName;  
        canConduct = true;  
    }  
}
```

MIXINS

```

mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}

```

MIXINS

```
class Musician {  
    // ...  
}  
mixin MusicalPerformer on Musician {  
    // ...  
}  
class SingerDancer extends Musician with MusicalPerformer {  
    // ...  
}
```

SUMMARY

SUMMARY

- ▶ OOP is real power to build amazing software

SUMMARY

- ▶ OOP is real power to build amazing software
- ▶ OOP is not only about classes and objects

SUMMARY

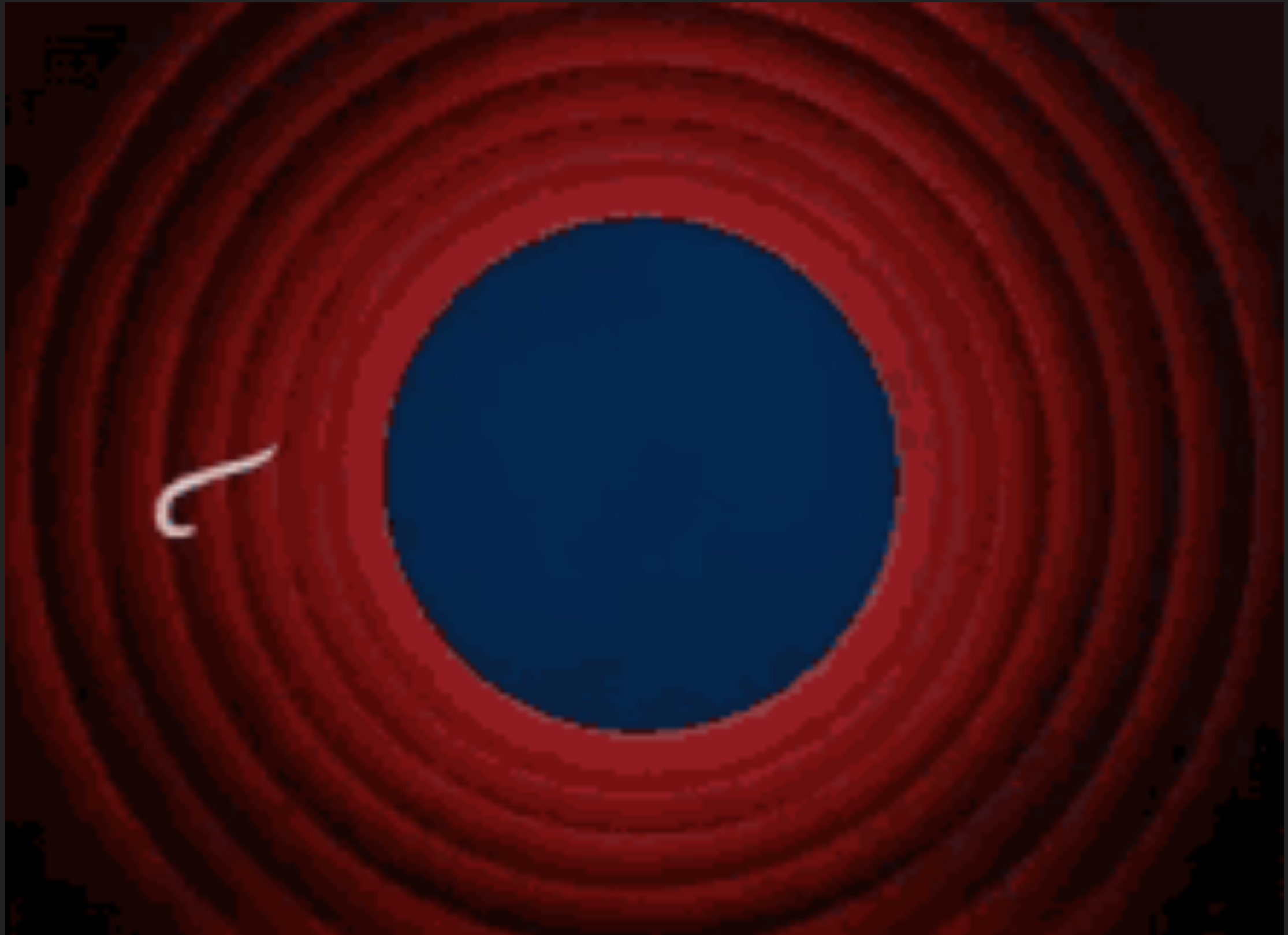
- ▶ OOP is real power to build amazing software
- ▶ OOP is not only about classes and objects
- ▶ Encapsulation is powerful tool to control access and give classes only things they need

SUMMARY

- ▶ OOP is real power to build amazing software
- ▶ OOP is not only about classes and objects
- ▶ Encapsulation is powerful tool to control access and give classes only things they need
- ▶ Inheritance doesn't work for big hierarchies - only for base class scenario or overrides

SUMMARY

- ▶ OOP is real power to build amazing software
- ▶ OOP is not only about classes and objects
- ▶ Encapsulation is powerful tool to control access and give classes only things they need
- ▶ Inheritance doesn't work for big hierarchies - only for base class scenario or overrides
- ▶ With interfaces, abstract classes and mixins we can do a lot of polymorphism magic





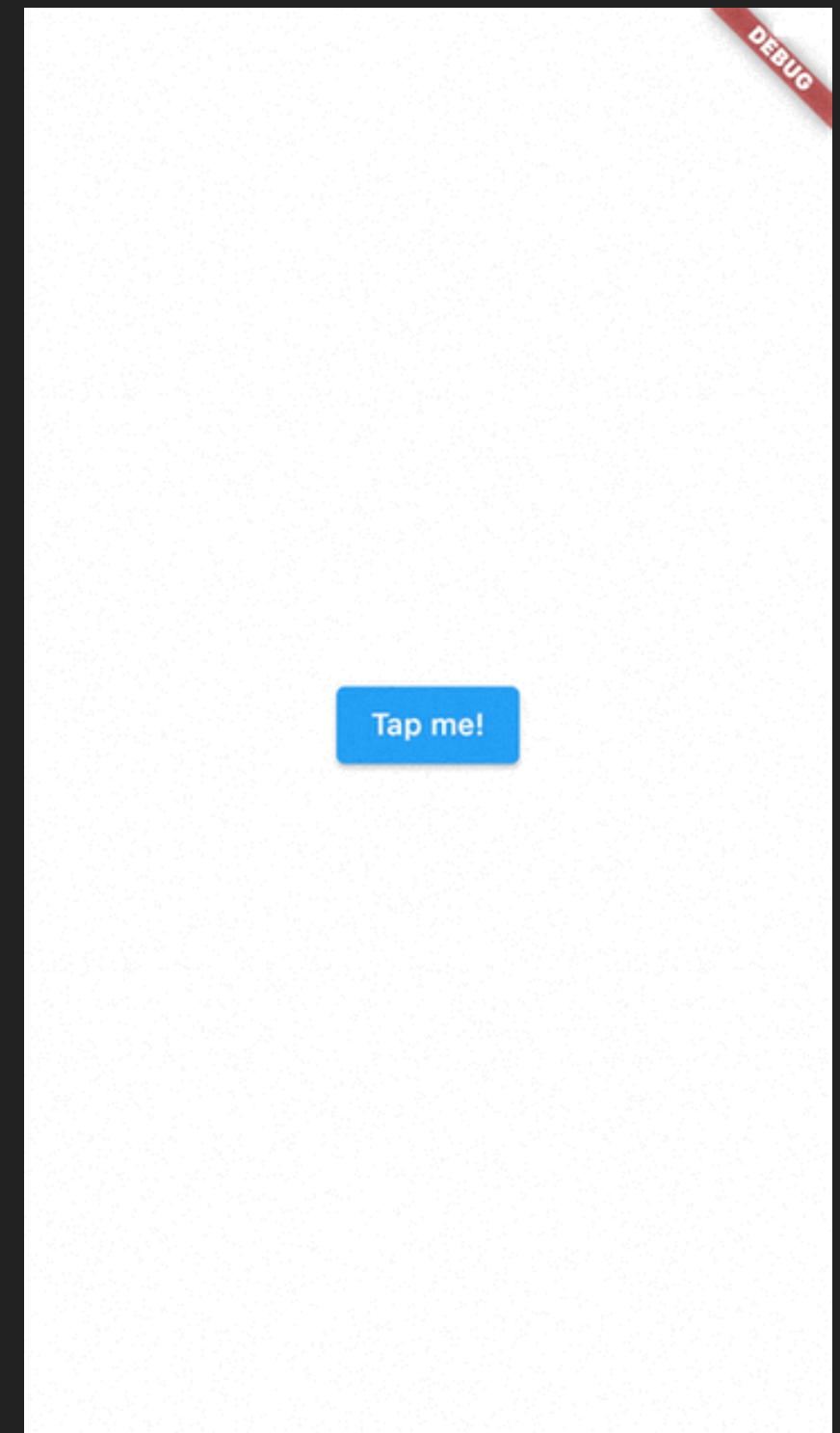
1. HOW TO CREATE GRADIENT

```
— body: Container(  
  decoration: const BoxDecoration(  
    gradient: LinearGradient(  
      begin: Alignment.topLeft,  
|     end: Alignment.bottomRight,  
      colors: [  
        Colors.yellowAccent,  
        Colors.redAccent,  
      ],  
    ), // LinearGradient  
  ), // BoxDecoration  
) // Container
```



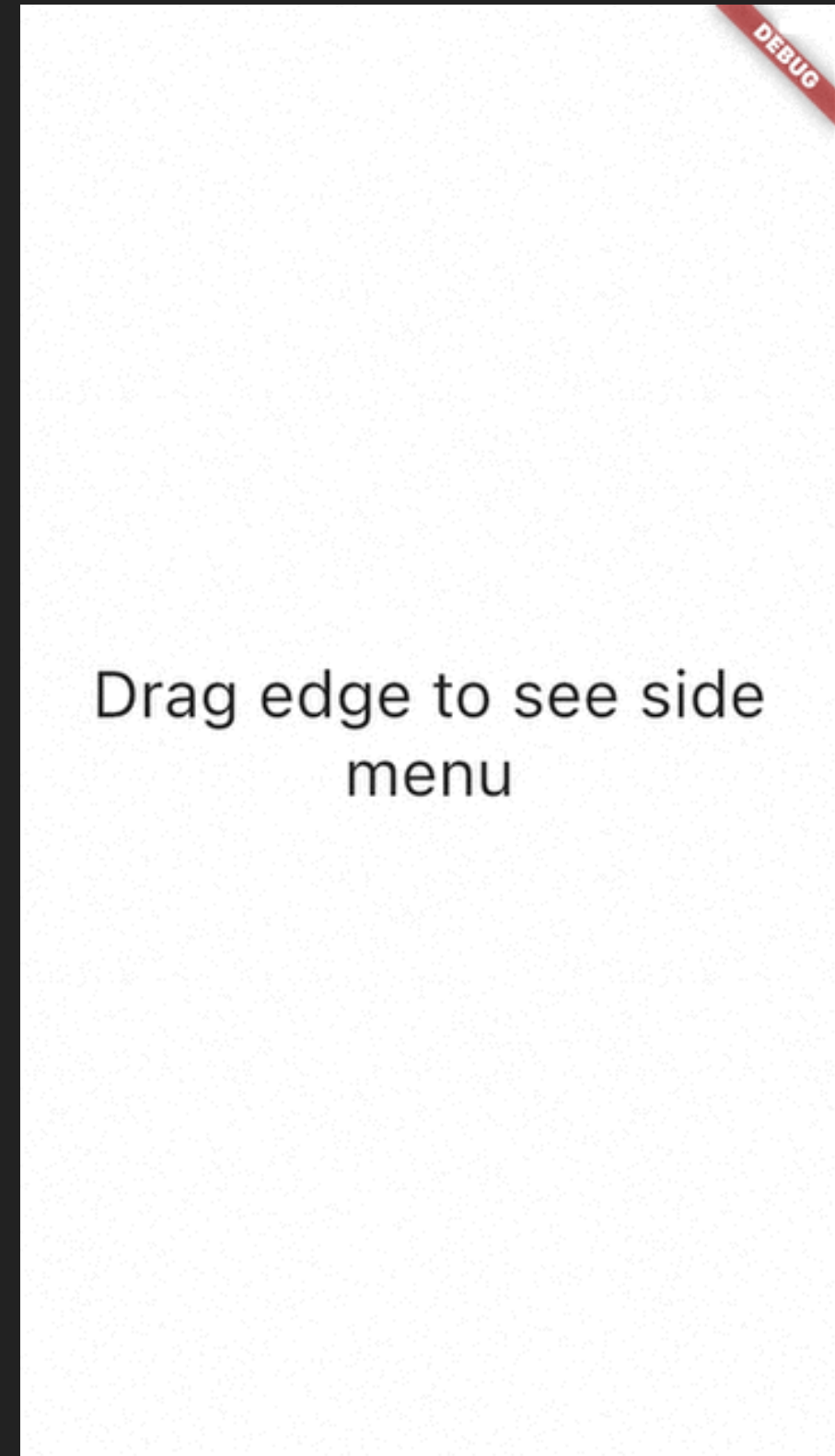
2. HOW TO SHOW DIALOG

```
showDialog(  
  context: context,  
  builder: (builder) {  
    var okButton = TextButton(  
      child: const Text("Dismiss me"),  
      onPressed: () {  
        Navigator.of(context).pop();  
      },  
    ); // TextButton  
  
    return AlertDialog(  
      title: const Text("Hello there!"),  
      actions: [okButton],  
    ); // AlertDialog  
  },  
);
```



3. HOW TO SHOW SIDE MENU

```
return Scaffold(  
  drawer: Drawer(  
    child: ListView(  
      children: const [  
        ListTile(  
          title: Text('User settings'),  
          leading: Icon(Icons.settings),  
        ), // ListTile  
        ListTile(  
          title: Text('Leave feedback'),  
          leading: Icon(Icons.feedback),  
        ), // ListTile  
        ListTile(  
          title: Text('About us'),  
          leading: Icon(Icons.info),  
        ), // ListTile  
      ],  
    ), // ListView  
  ), // Drawer  
);
```



Drag edge to see side
menu