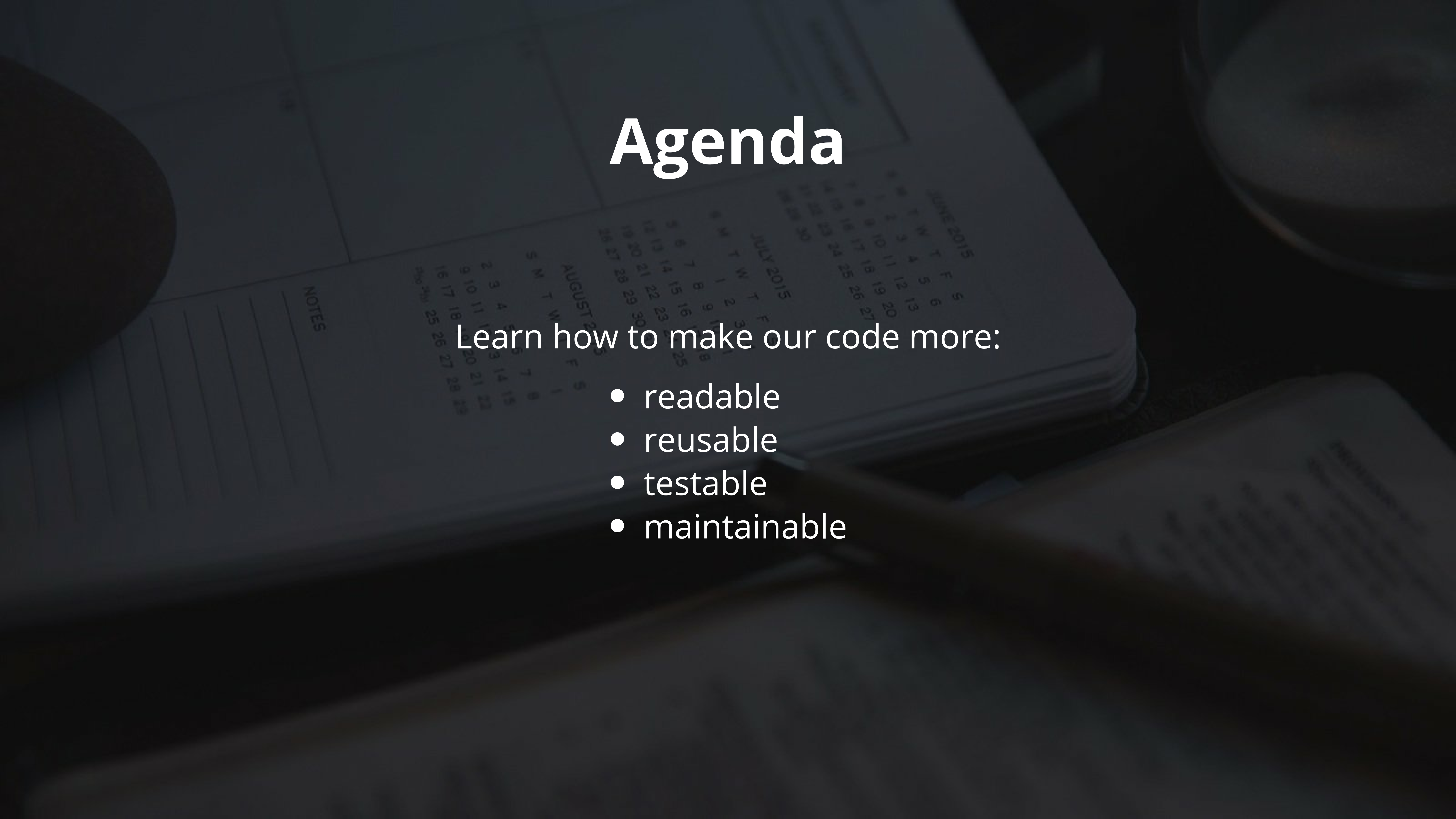
Object Calisthenics

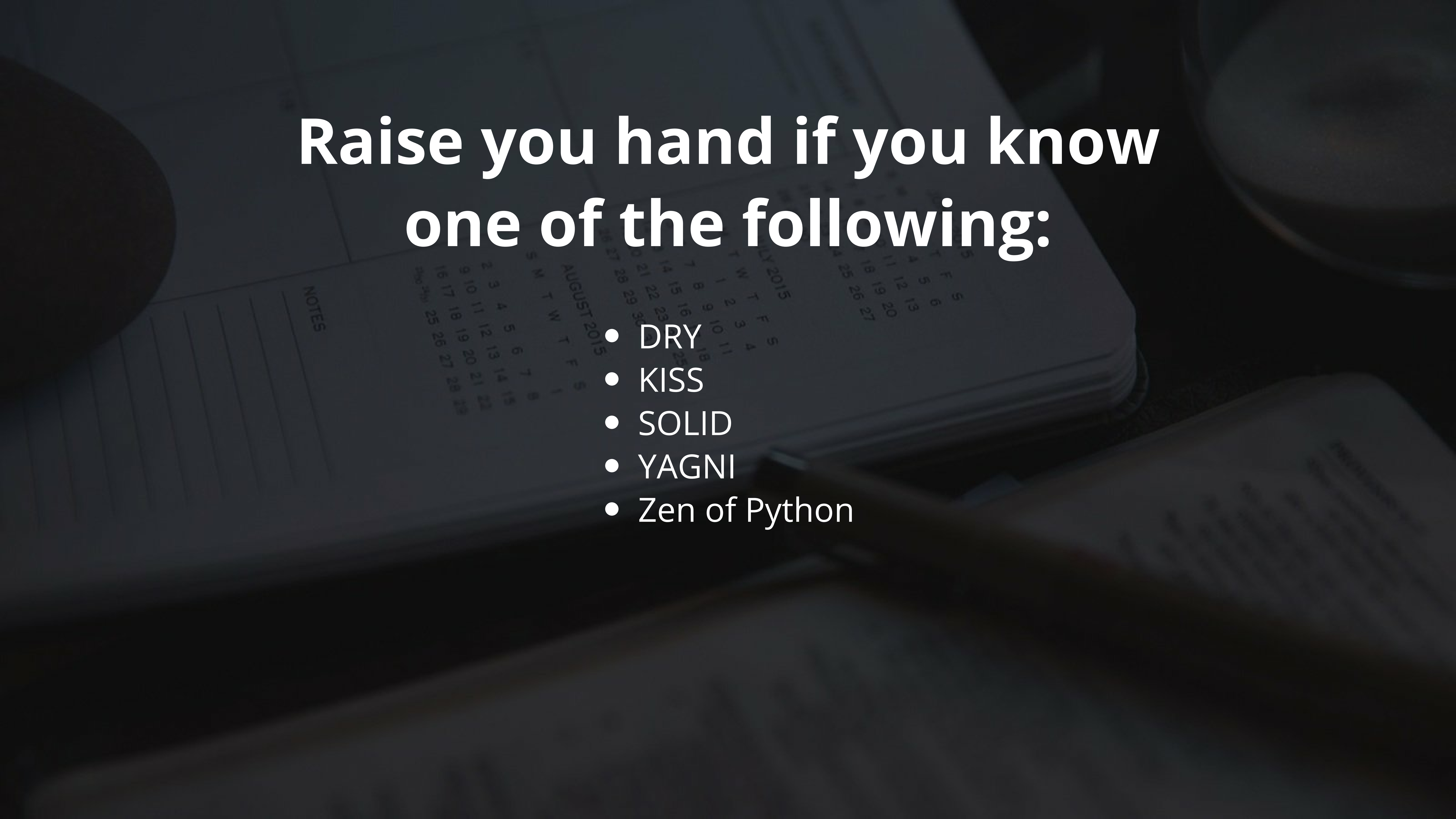
9 steps to better OO code



Agenda

Learn how to make our code more:

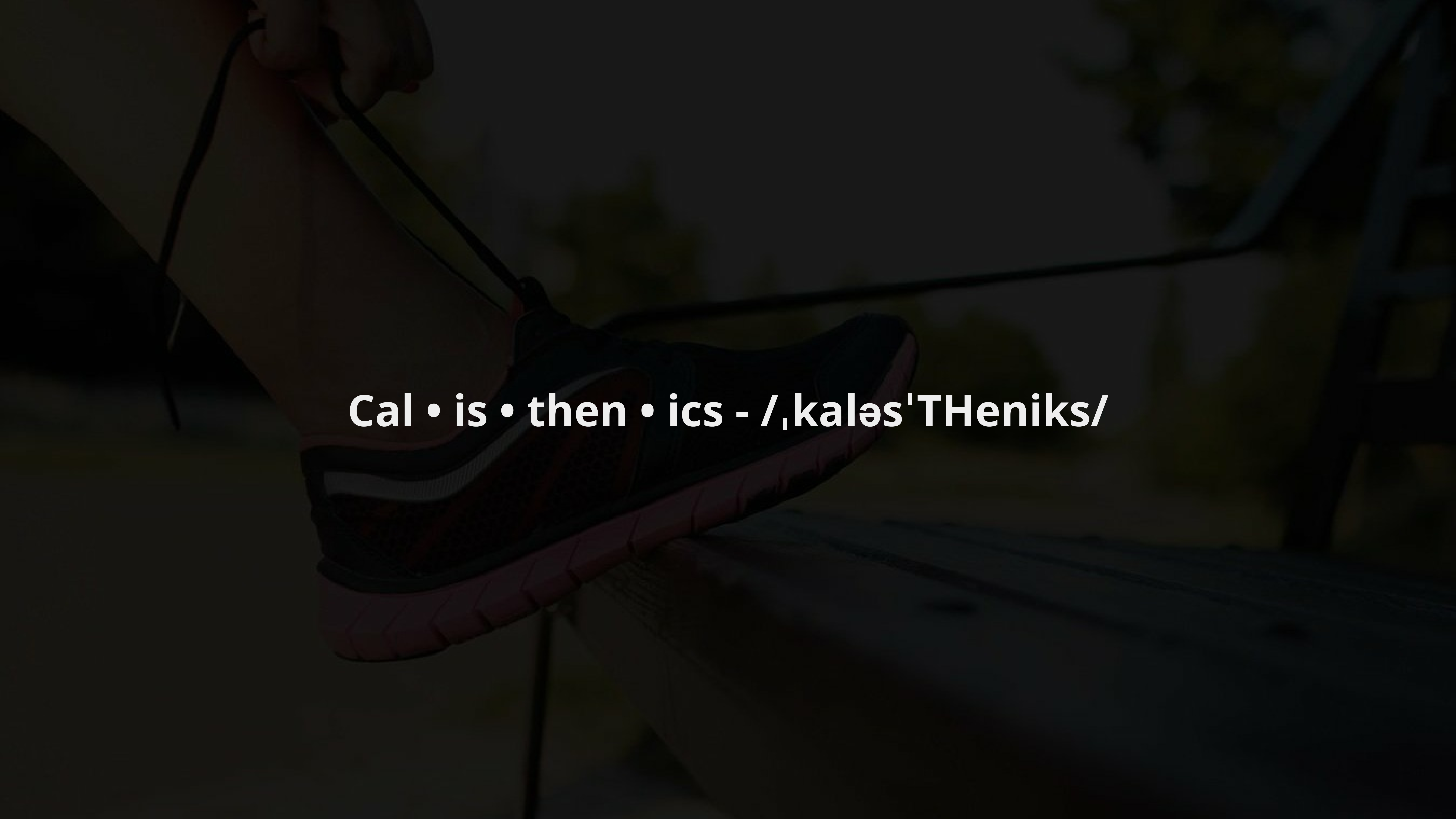
- readable
- reusable
- testable
- maintainable



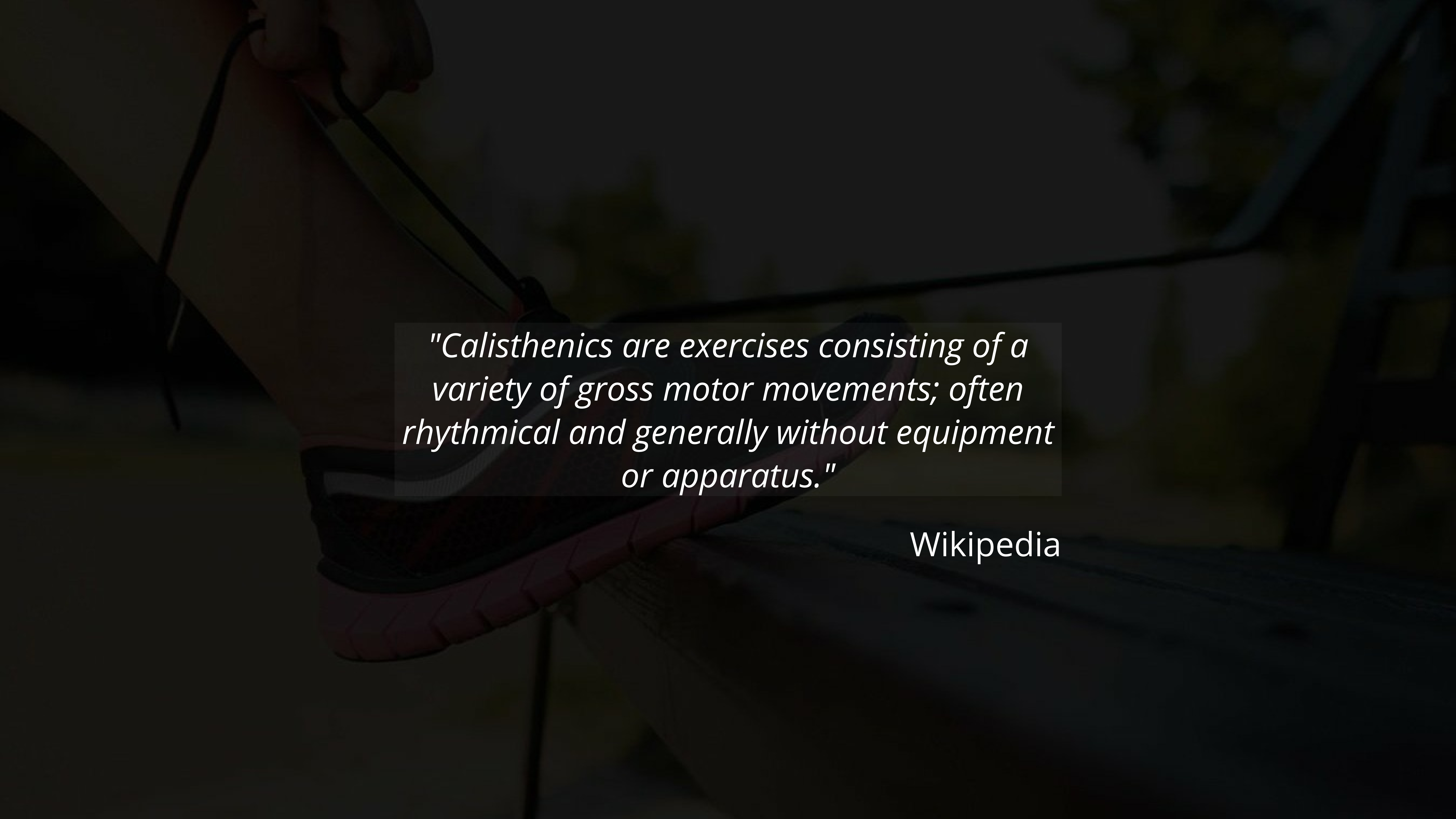
Raise you hand if you know one of the following:

- DRY
- KISS
- SOLID
- YAGNI
- Zen of Python

Calisthenics



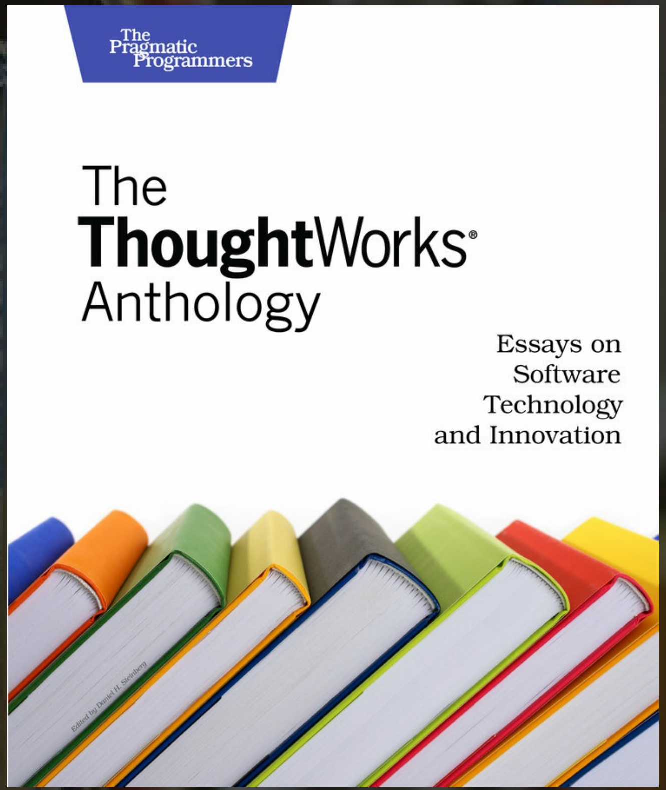
Cal • is • then • ics - /ˌkaləsˈTHeniks/



"Calisthenics are exercises consisting of a variety of gross motor movements; often rhythmical and generally without equipment or apparatus."

Wikipedia

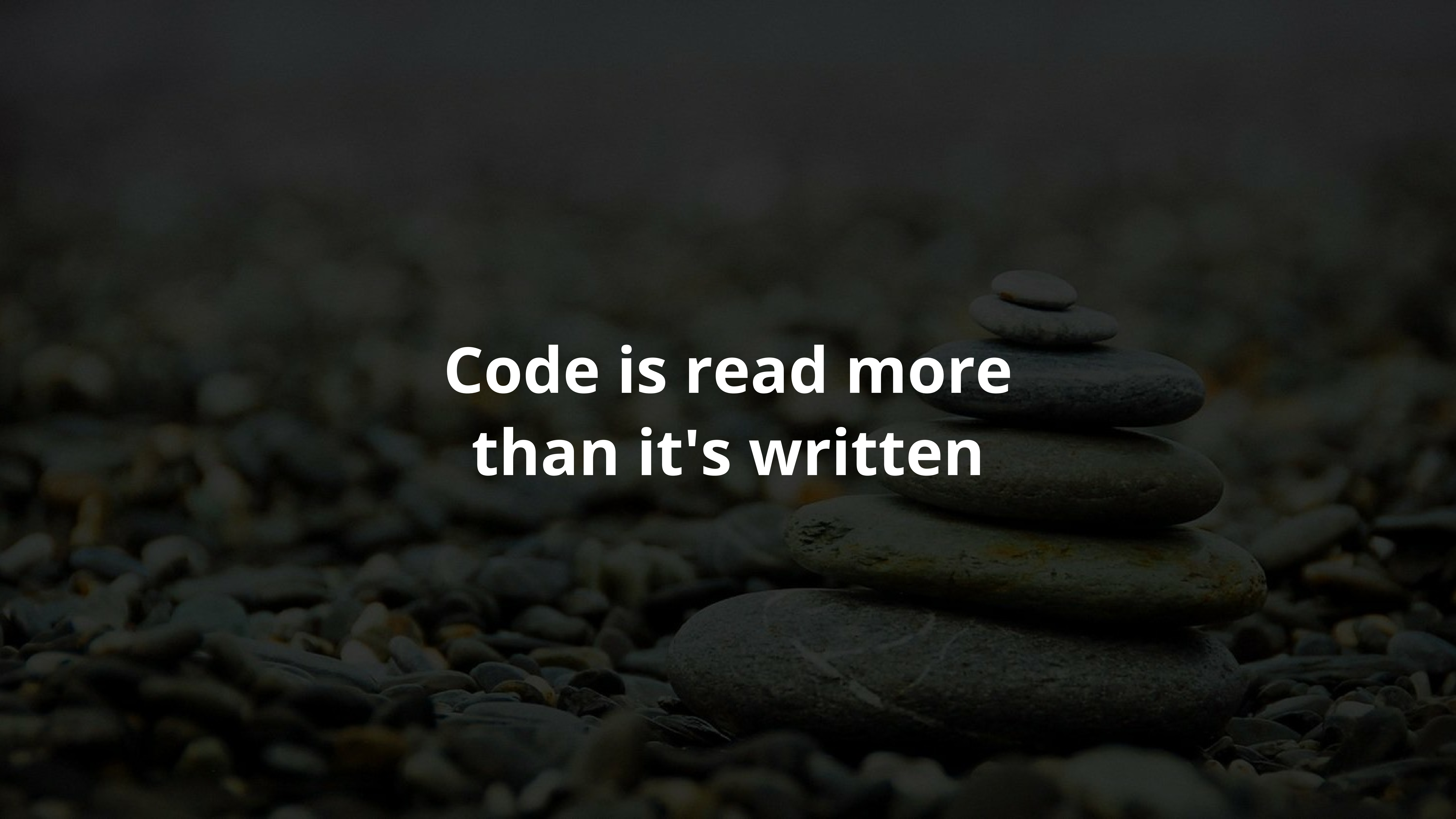
Object Calisthenics


The Pragmatic Programmers
The ThoughtWorks® Anthology
Essays on Software Technology and Innovation

Jeff Bay

Written for Java

Why bother?



Code is read more than it's written

Rule #1

Only one level of indentation per method

```
class Board(object):
    def __init__(self, data):
        # Level 0
        self.buf = ""
        for i in range(10):
            # Level 1
            for j in range(10):
                # Level 2
                self.buf += data[i][j]
```

```python
class Board(object):
    def __init__(self, data):
        self.buf = ""
        self.collect_rows(data)

    def collect_rows(self, data):
        for i in range(10):
            self.collect_row(data[i])

    def collect_row(self, row):
        for j in range(10):
            self.buf += row[j]
```

Benefits

- Single responsibility
- Better naming
- Shorter methods
- Reusable methods

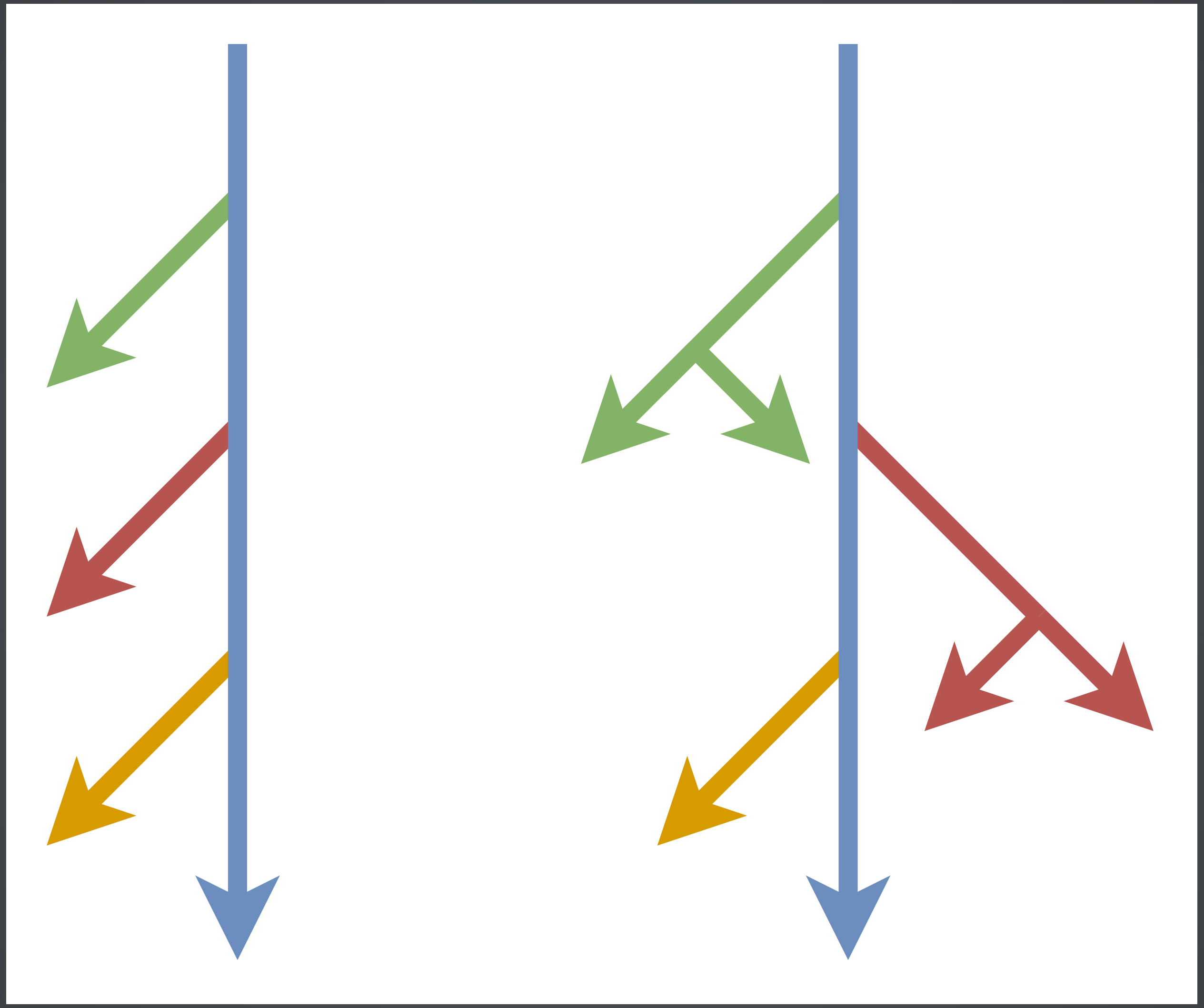
Rule #2

Do not use *else* keyword

```python
if options.getCategories() is None:
    ...
elif len(options.getCategories()) == 1:
    ...
elif SPECIAL_CATEGORY in options.getCategories():
    ...
elif options.getCategories() and options.getQuery():
    ...
elif options.getContentType():
    ...
```

```
def login (self, request):
    if request.user.is_authenticated():
        return redirect("homepage")
    else:
        messages.add_message(request,
                             messages.INFO,
                             'Bad credentials')
        return redirect("login")
```

```
def login (self, request):
    if request.user.is_authenticated():
        return redirect("homepage")

    messages.add_message(request,
                         messages.INFO,
                         'Bad credentials')
    return redirect("login")
```



Extract code

Default value

Polymorphism

Strategy pattern

State pattern

Benefits

- Avoids code duplication
- Lower complexity
- Readability

Rule #3

Wrap primitive types if it has behaviour

Value Object in DDD

```python
class Validator(object):
    def check_date(self, year, month, day):
        pass

# 10th of December or 12th of October?
validator = Validator()
validator.check_date(2016, 10, 12)
```

```
class Validator(object):
    def check_date(year: Year, month: Month, day: Day) -> bool:
        pass

# Function call leaves no doubt.
validator.check_date(Year(2016), Month(10), Day(12))
```

Benefits

- Encapsulation
- Type hinting
- Attracts similar behaviour

Rule #4

Only one dot per line

OK: Fluent interface

```python
class Poem(object):
    def __init__(self, content):
        self.content = content

    def indent(self, spaces):
        self.content = " " * spaces + self.content
        return self

    def suffix(self, content):
        self.content = self.content + " - " + content
        return self

Poem("Road Not Travelled").indent(4)\
    .suffix("Robert Frost").content
```

Not OK: getter chain

```python
class CartService(object):
    def get_token(self):
        token = self.get_service('auth')\
            .auth_user('user', 'password')\
            .get_result()\
            .get_token()

        return token

# 1. What if None is returned instead of object?
# 2. How about exceptions handling?
```

```python
class Location(object):
    def __init__(self):
        self.current = Piece()

class Piece(object):
    def __init__(self):
        self.representation = " "

class Board(object):
    def board_representation(self, board):
        buf = ''
        for field in board:
            buf += field.current.representation

        return buf
```

```python
class Location(object):
    def __init__(self):
        self.current = Piece()

    def add_to(self, buffer):
        return self.current.add_to(buffer)

class Piece(object):
    def __init__(self):
        self.representation = " "

    def add_to(self, buffer):
        return buffer + self.representation

class Board(object):
    def board_representation(self, board):
        buf = ''
        for field in board:
            buf = field.add_to(buf)

        return buf
```

Benefits

- Encapsulation
- Demeter's law
- Open/Closed Principle

Rule #5

Do not abbreviate

Why abbreviate?

Too many responsibilities

Name too long?

Split & extract

Duplicated code?

Refactor!

Benefits

- Clear intentions
- Indicate underlying problems

Rule #6

Keep your classes small

What is small class?

- 15-20 lines per method
- 50 lines per class
- 10 classes per module

Benefits

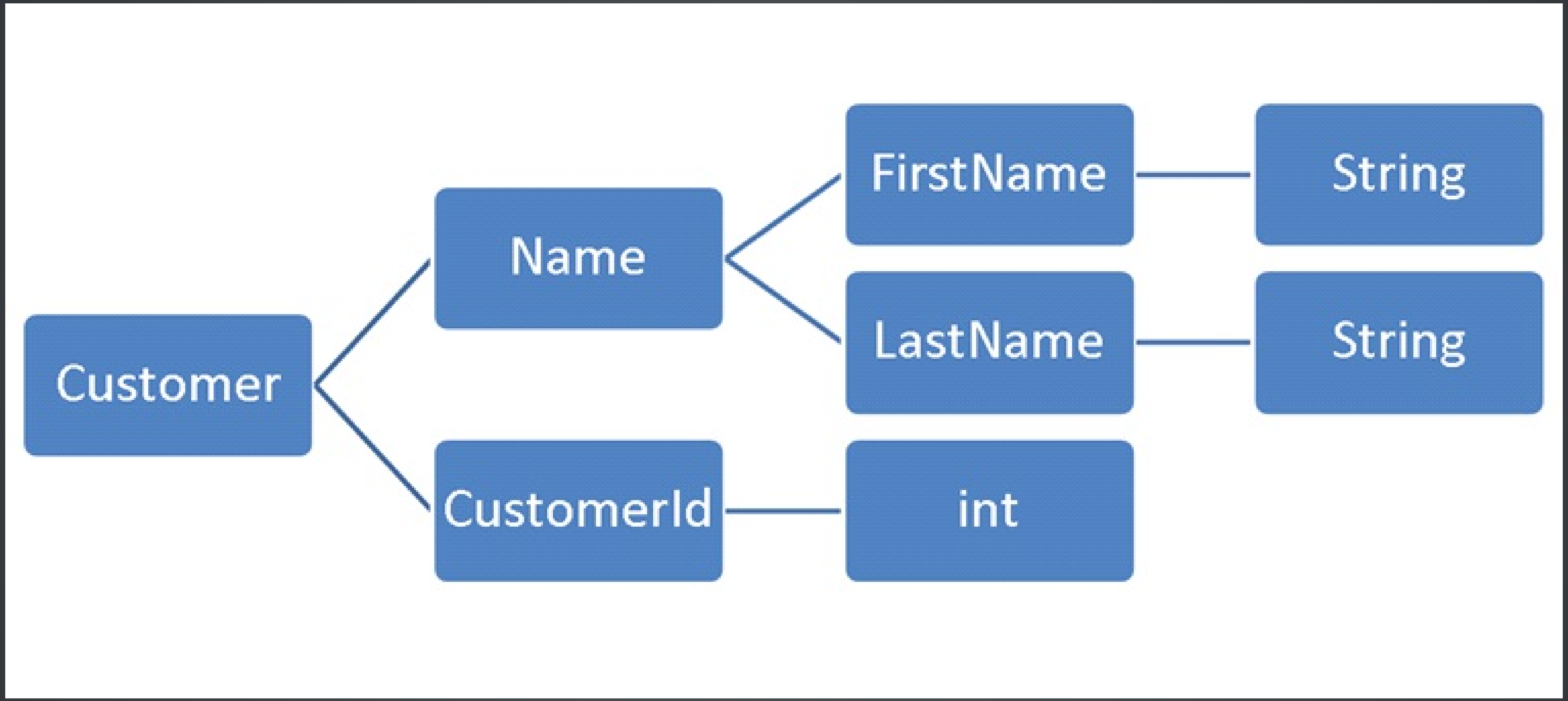
- Single Responsibility
- Smaller modules

Rule #7

No more than 2 instance variable per class

Class should handle single variable state

In some cases it might be two variables


Customer
Name
CustomerId
FirstName
LastName
int
String
String

```
class CartService(object):
    def __init__(self):
        self.logger = Logger()
        self.cart = CartCollection()
        self.translationService = TranslationService()
        self.authService = AuthService()
        self.userService = UserService()
```

Benefits

- High cohesion
- Encapsulation
- Fewer dependencies

Rule #8

First class collections

collections module

Benefits

- Single Responsibility

Rule #9

Do not use setters/getters

Accessors are fine

Don't make decisions outside of class

Let class do it's job

Tell, don't ask

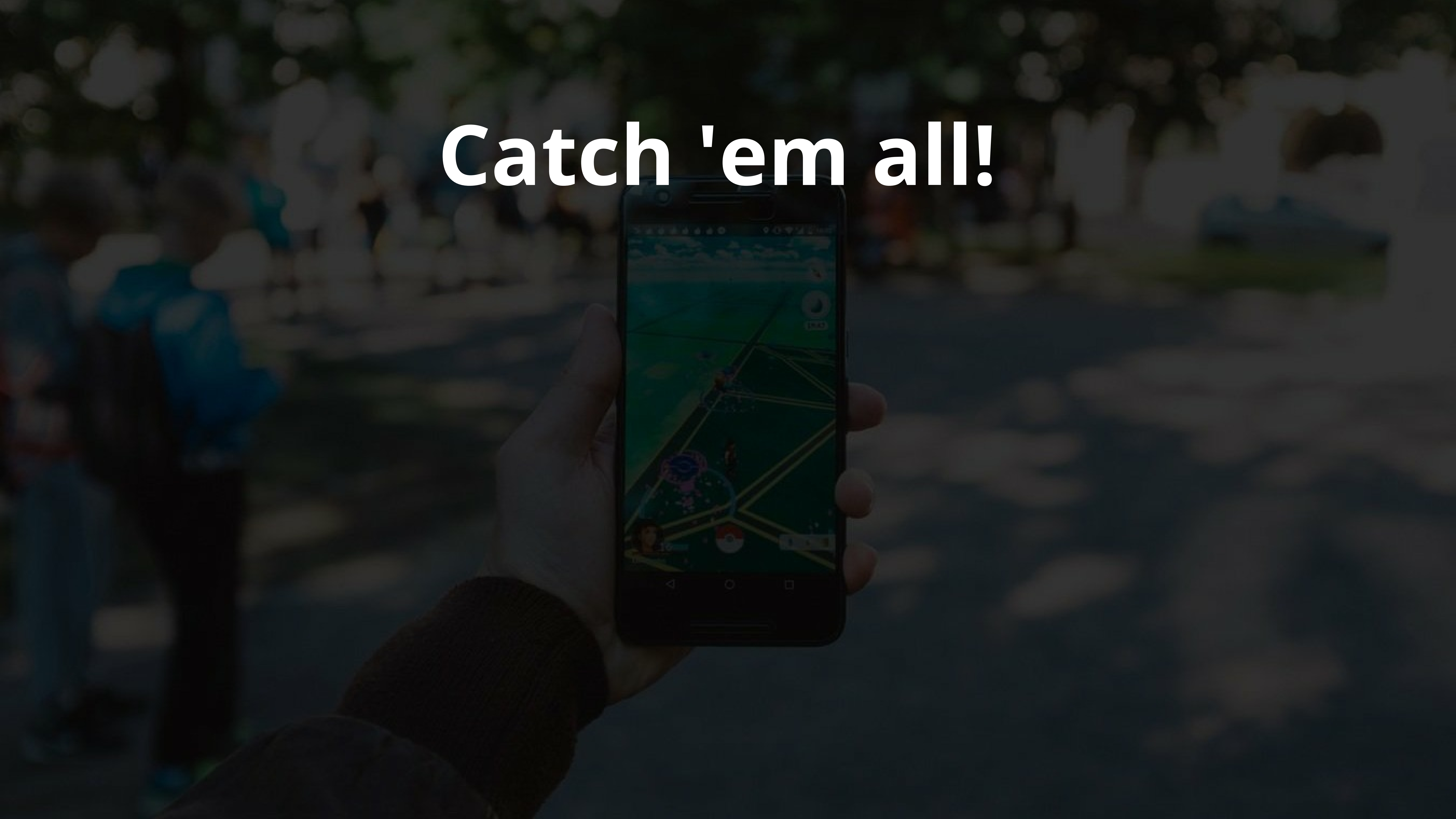
```python
class Game(object):
    def __init__(self):
        self.score = 0

    def set_score(self, score):
        self.score = score

    def get_score(self):
        return self.score

# Usage
ENEMY_DESTROYED_SCORE = 10
game = Game()
game.set_score(game.get_score() + ENEMY_DESTROYED_SCORE)
```

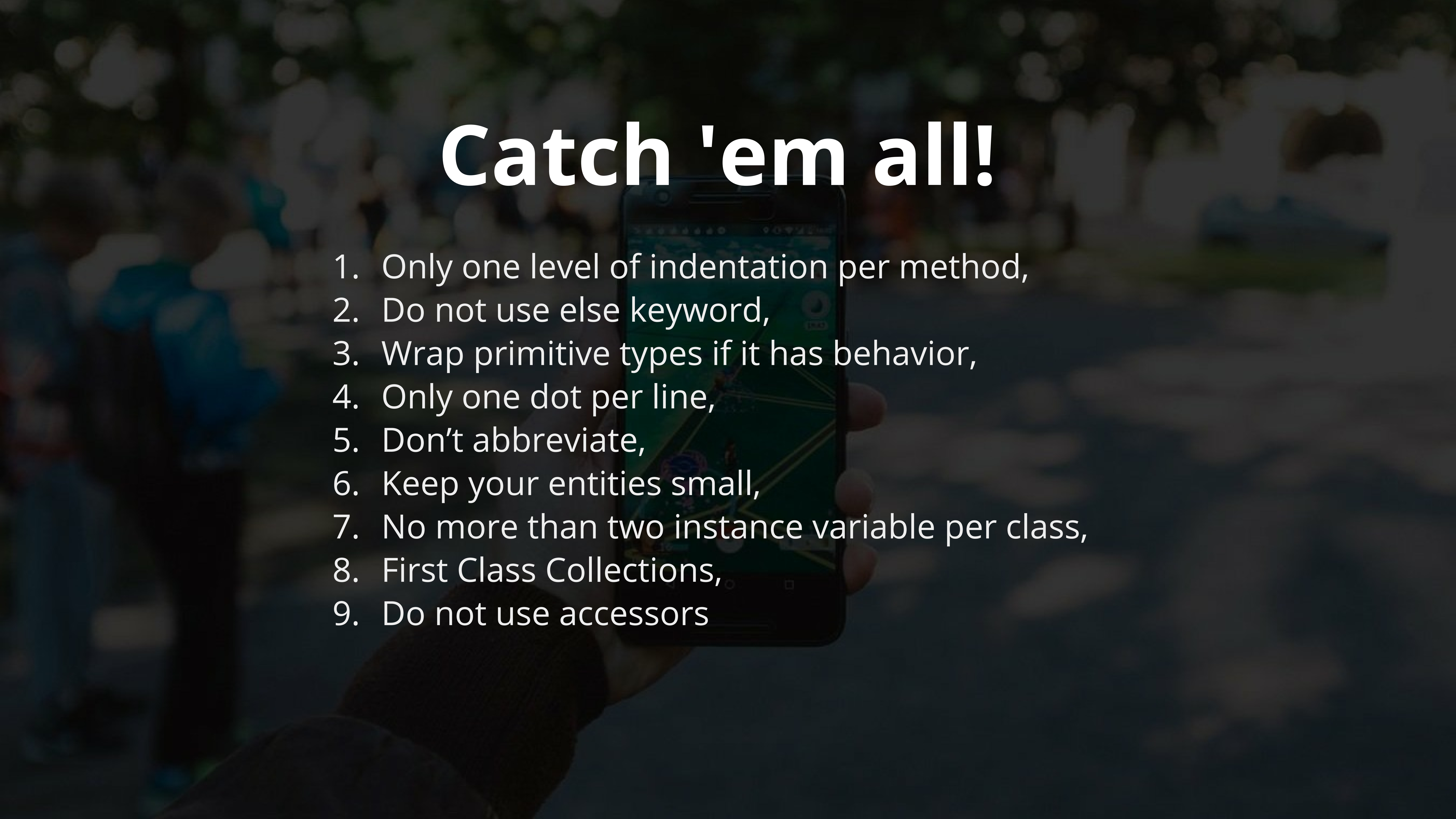
```
class Game(object):
    def __init__(self):
        self.score = 0

    def add_score(self, score):
        self.score += score

# Usage
ENEMY_DESTROYED_SCORE = 10
game = Game()
game.add_score(ENEMY_DESTROYED_SCORE)
```

Benefits

- Open/Closed Principle

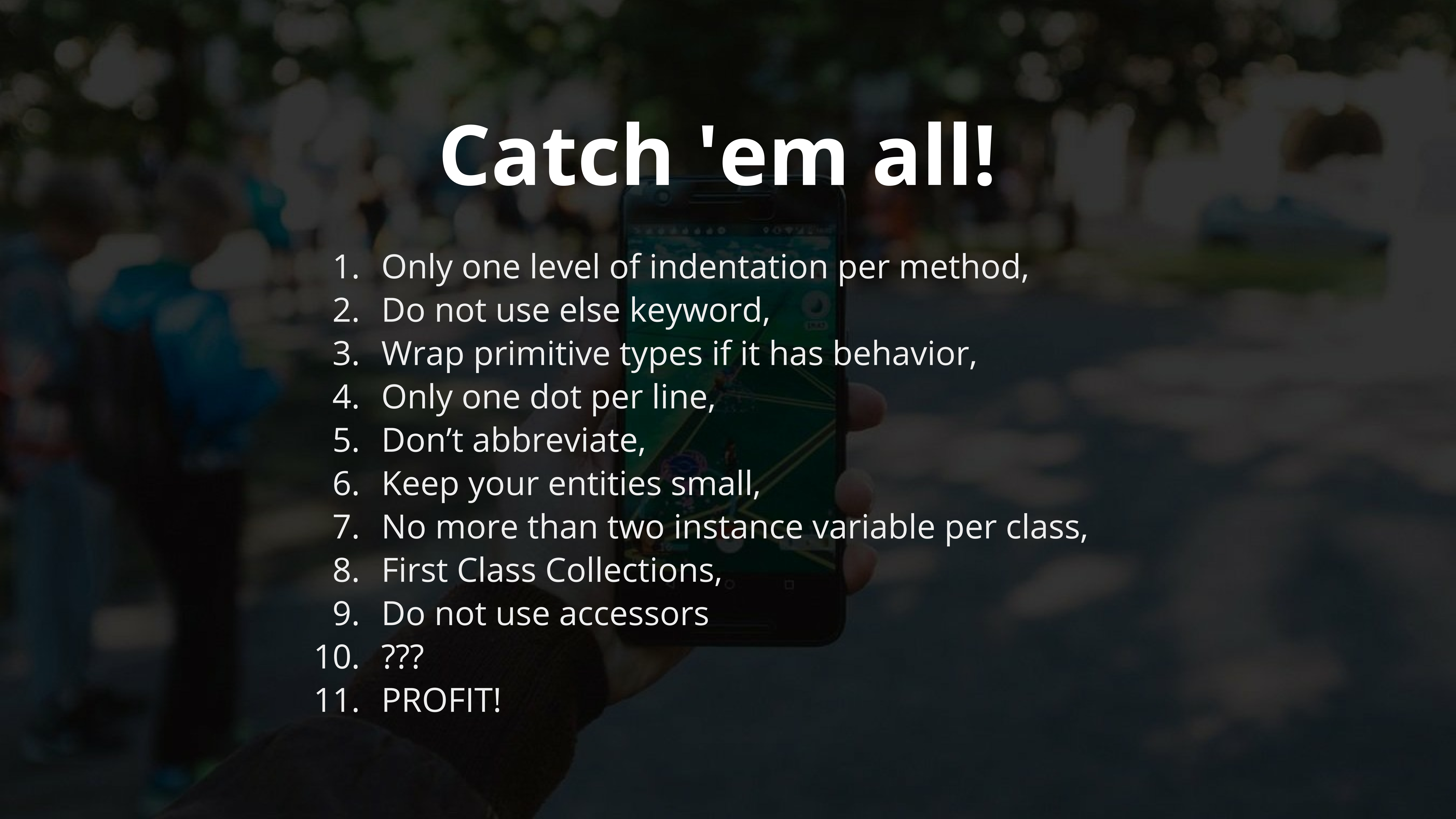
Catch 'em all!





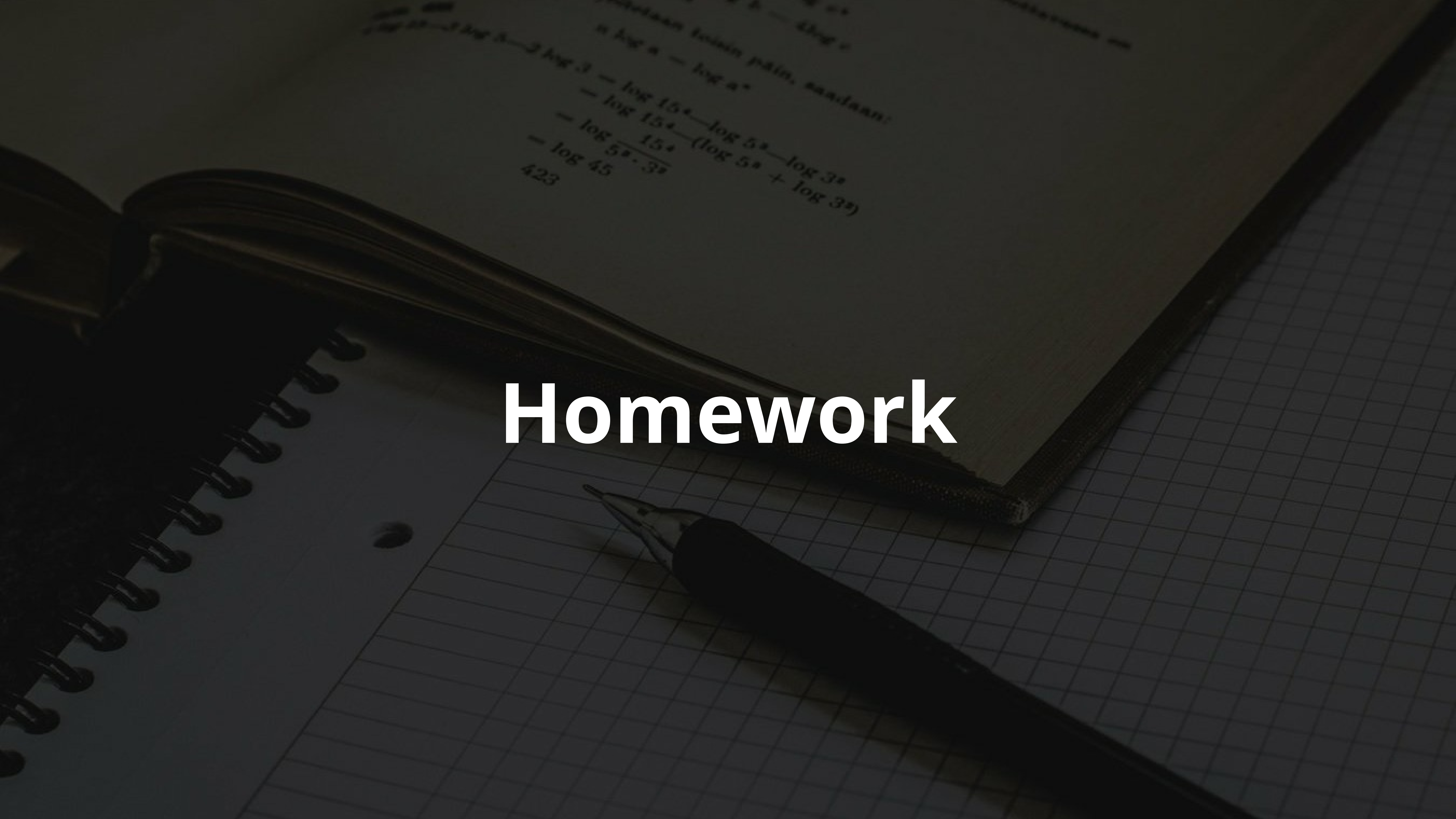
Catch 'em all!

1. Only one level of indentation per method,
2. Do not use else keyword,
3. Wrap primitive types if it has behavior,
4. Only one dot per line,
5. Don’t abbreviate,
6. Keep your entities small,
7. No more than two instance variable per class,
8. First Class Collections,
9. Do not use accessors

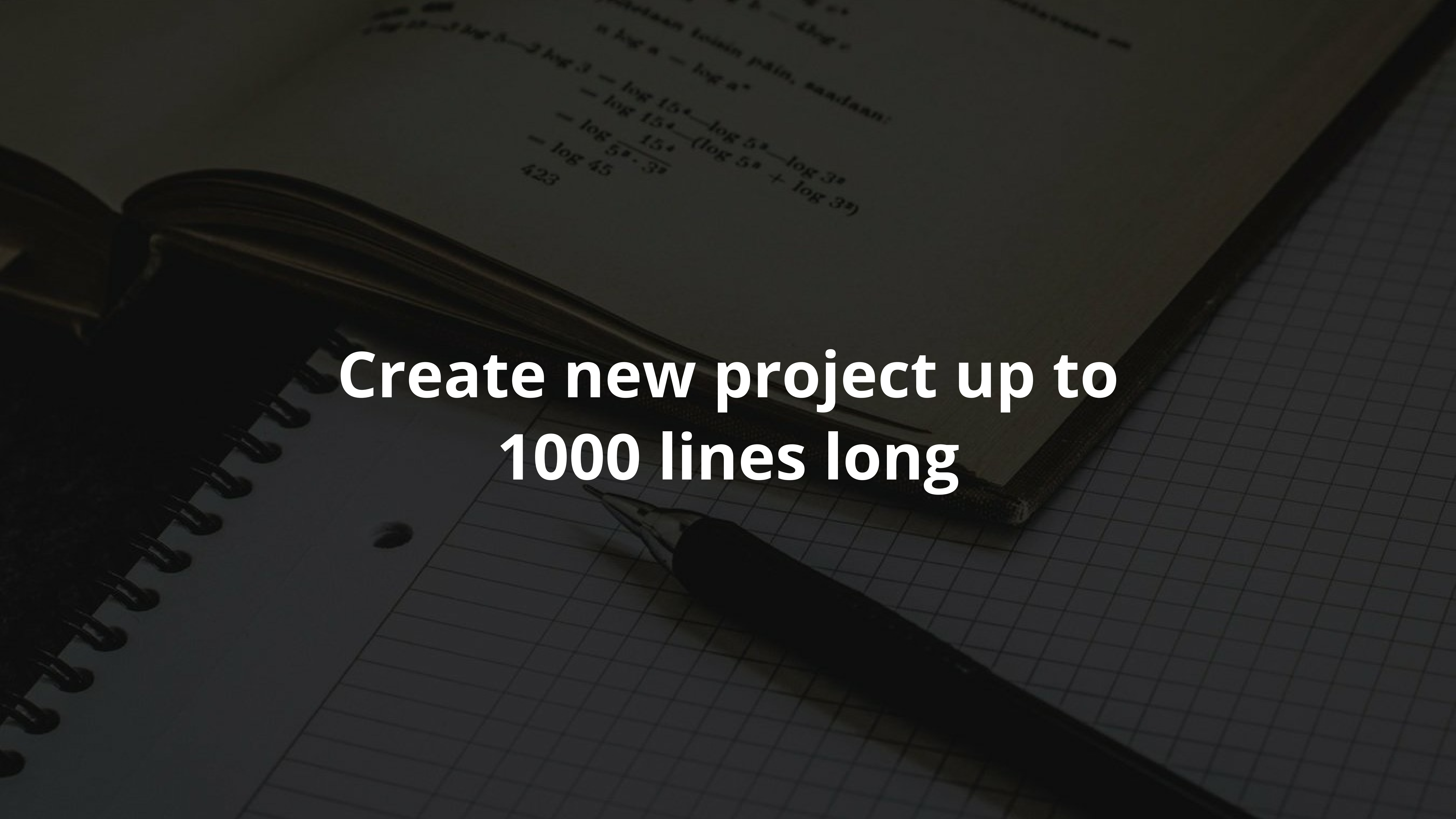


Catch 'em all!

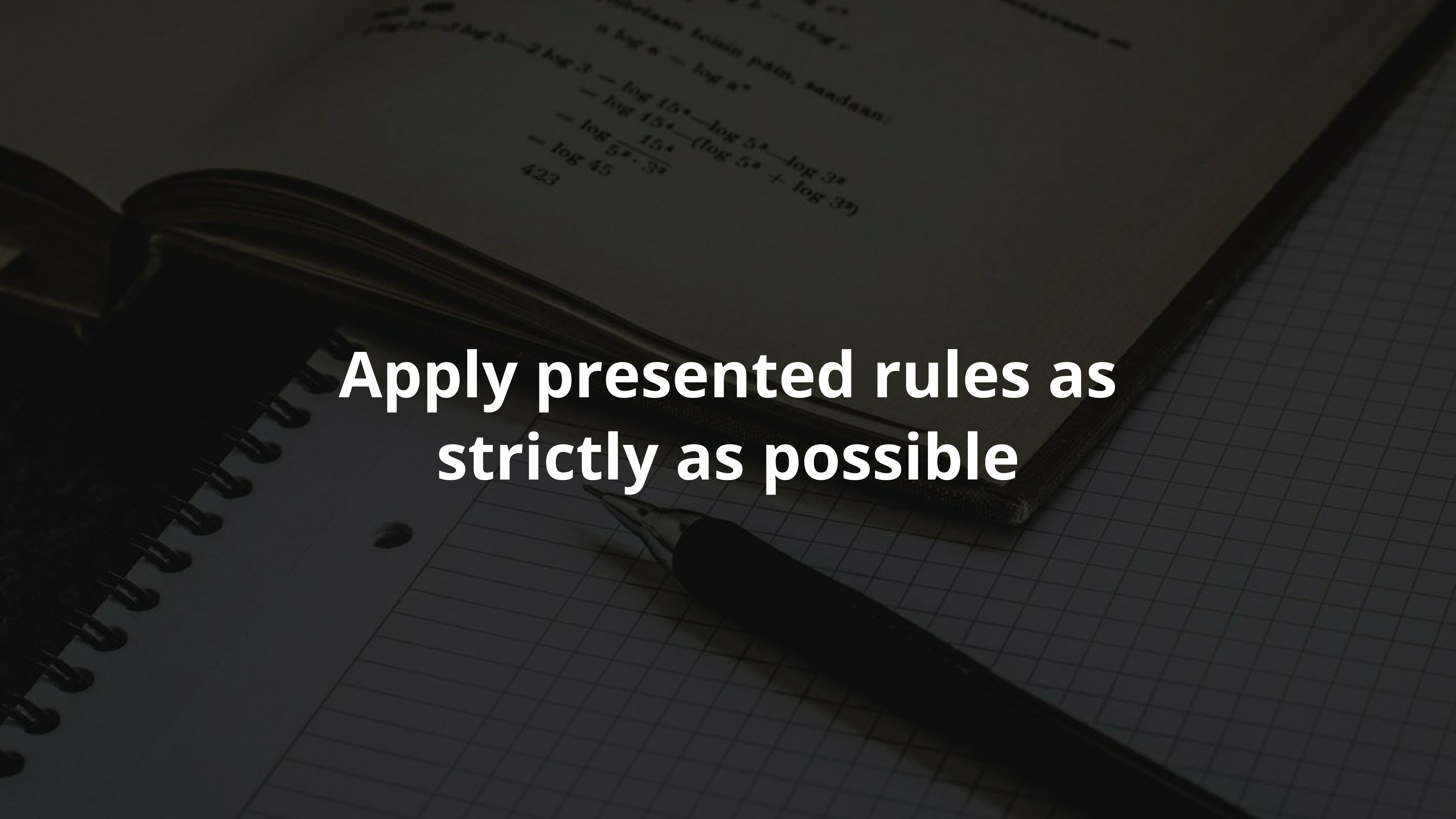
1. Only one level of indentation per method,
2. Do not use else keyword,
3. Wrap primitive types if it has behavior,
4. Only one dot per line,
5. Don’t abbreviate,
6. Keep your entities small,
7. No more than two instance variable per class,
8. First Class Collections,
9. Do not use accessors
10. ???
11. PROFIT!



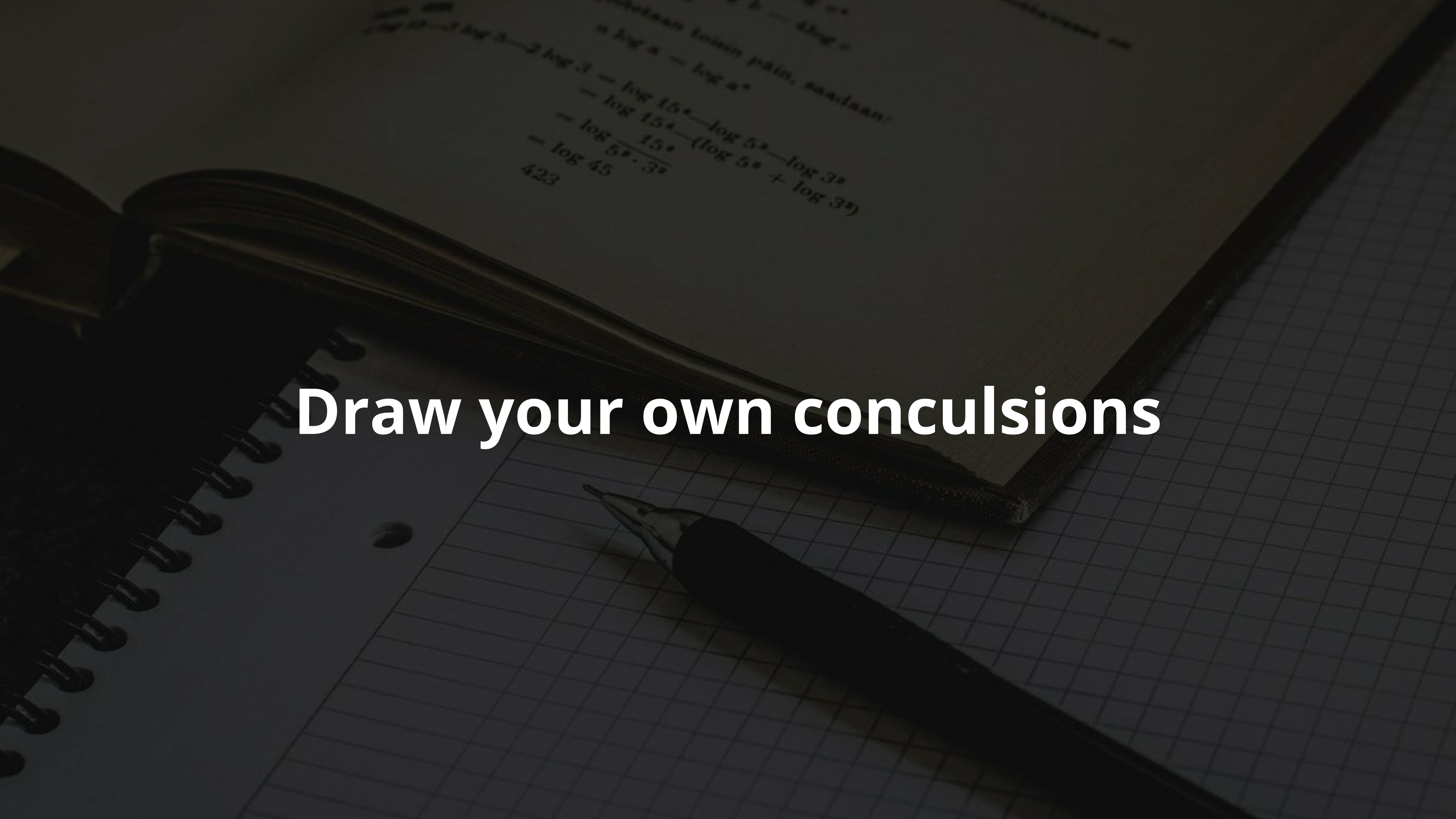
Homework



Create new project up to 1000 lines long



Apply presented rules as strictly as possible



Draw your own conculsions

Customize these rules

Final thoughts

These are not best practices

These are just guidelines

Use with caution!

Questions?

Thank you!