
SIGMA
Software


SIGNL4
Software
UNIVERSITY


SIGMA
Software
UNIVERSITY

ЛЕКЦІЯ 8

# “Класи”

9

SIGMA SOFTWARE UNIVERSITY

# Класи

Класи

Методи та індексатори

Ініціалізація

Деініціалізація

Наслідування

Optional Chaining

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Класи

## Класи

Методи та індексатори

Ініціалізація

Деініціалізація

Наслідування

Optional Chaining

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Класи

Клас - це *іменовий* тип загального призначення, що представляє гнучку конструкцію, яка містить дані деяких типів та визначає поведінку роботи над ними, а також служить у якості блока коду при побудові програм.

Клас - це *тип за посиланням*.

Елементи класу, які представляють дані, називається *властивостями* класу.

Елементи класу, які представляють інтерфейс для роботи з даними, що містяться у ньому, називаються *методами* класу.

# Класи

OBJ-C

Objective-C визначає наступний синтаксис опису класу:

```
@interface SimpleClass : NSObject
   //properties
   //methods
   //initializations
@end
```

SIGMA SOFTWARE UNIVERSITY

# Класи

Swift визначає наступний синтаксис опису класу:

```
accessLevel class ClassName [ : ParentClassName ] , Protocol1, … {
// let or var declaration(s)
// init definition(s)
// func definition(s)
// subscript definition(s)
}
```

accessLevel - рівень доступу (public, internal, private)

# Класи: екземпляр

Клас - це тип даних, а сутність типу класа називається її *екземпляром* або об’єктом.

Swift надає наступний синтаксис оголошення та створення екземпляра класу.

let constantInstanceName: ClassName = ClassName(…)
var variableInstanceName: ClassName = ClassName(…)

# Класи

```
3  import UIKit
4
5  final public class Car {
6
7      private var modelName:String
8      private var maximumSpeed:Int
9      private var price:Int
10
11     init(name:String, speed:Int, price:Int) {
12         modelName = name
13         maximumSpeed = speed
14         self.price = price
15     }
16
17     var description:String {
           return "Car named \(modelName) with speed \(maximumSpeed) cost \(price)"
19     }
20 }
21
   let carExample = Car(name:"Awesome car", speed:100, price:1000)
   print(carExample.description)
```

```
"Car named Awesome c...
Car
"Car named Awesome c...
```

SIGMA SOFTWARE UNIVERSITY

# Класи

OBJ-C

```objc
#import <Foundation/Foundation.h>

@interface Car : NSObject

@property (nonatomic, copy) NSString *name;
@property (assign, nonatomic) NSInteger speed;
@property (assign, nonatomic) NSInteger price;

- (instancetype)init:(NSString *)carName speed:(NSInteger)carSpeed price:(NSInteger)carPrice;

@end
```

```objc
#import "Car.h"

@implementation Car

#pragma mark - LifeCycle

- (instancetype)init:(NSString *)carName speed:(NSInteger)carSpeed price:(NSInteger)carPrice
{
    self = [super init];
    if (self) {
        _name = carName;
        _speed = carSpeed;
        _price = carPrice;
    }
    return self;

}

#pragma mark - Debug

- (NSString *)description
{
    return [[NSString alloc] initWithFormat:@"Car named %@, with speed @li, cost %li", _name,
        (long)_speed, (long)_price ];
}

@end
```

```objc
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *newCar = [[Car alloc] init:@"Awesome" speed:100 price:1000];
        NSLog(@"%@", newCar.description);
    }
    return 0;
}
```

SIGMA SOFTWARE UNIVERSITY

# Класи

## Класи

Методи та індексатори

Ініціалізація

Деініціалізація

Наслідування

Optional Chaining

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Класи

Класи

**Методи та індексатори**

Ініціалізація

Деініціалізація

Наслідування

Optional Chaining

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Методи

Методи - це функції, які дозволяють виконувати операції над властивостями.

Методи поділяються на методи екземпляра та методи типа.

Методи, які виконають операції над властивостями *екземпляра та(або) типу*, називається *методами екземпляра*.

Методи, які виконають операції над властивостями *типу*, називається *методами типу*.

self - це ключове слово для доступу до всіх властивостей та методів екземпляра.

self - це власне сам екземпляр.

# Методи: instance

```swift
import UIKit

final public class Car {

    var modelName:String
    public var maximumSpeed:Int
    private (set) var price:Int

    init(name:String, speed:Int, price:Int) {
        modelName = name
        maximumSpeed = speed
        self.price = price
    }

    var description:String {
        return "Car named \(modelName) with speed \(maximumSpeed) cost \(price)"
    }

    func speedUp() {
        maximumSpeed += 1;
    }
}

let carExample = Car(name:"Awesome car", speed:100, price:1000)
print(carExample.description)

carExample.speedUp()
carExample.maximumSpeed
```

```
"Car named Awesome c...
Car
"Car named Awesome c...
Car
101
```

SIGMA SOFTWARE UNIVERSITY

# Методи: instance

OBJ-C

```objc
//
//  Car.h
//  test
//
//  Created by Kirill Gorbushko on 04.03.18.
//  Copyright © 2018 - present. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface Car : NSObject

@property (nonatomic, copy) NSString *name;
@property (assign, nonatomic) NSInteger speed;
@property (assign, nonatomic) NSInteger price;

- (instancetype)init:(NSString *)carName speed:(NSInteger)carSpeed price:(NSInteger)carPrice;

- (void)speedUp;

@end
```

```objc
#pragma mark - LifeCycle

- (instancetype)init:(NSString *)carName speed:(NSInteger)carSpeed price:(NSInteger)carPrice
{
    self = [super init];
    if (self) {
        _name = carName;
        _speed = carSpeed;
        _price = carPrice;
    }
    return self;
}

#pragma mark - Public

- (void)speedUp
{
    self.speed++;
}

#pragma mark - Debug

- (NSString *)description
{
    return [[NSString alloc] initWithFormat:@"Car named %@, with speed @li, cost %li", _name, (long)_speed, (long)_price ];
}

@end
```

SIGMA SOFTWARE UNIVERSITY

# Методи: type

```
24
25      class func createBWM() -> Car {
            return Car(name:"BMW", speed:300, price:5000)
27      }
28  }
29
    let carExample = Car(name:"Awesome car", speed:100, price:1000)
    print(carExample.description)
32
    carExample.speedUp()
    carExample.maximumSpeed
35
    let bmw = Car.createBWM()
    print(bmw.description)
```

```
Car
Car
"Car named Awesome c...
Car
101
Car
"Car named BMW with...
```

SIGMA SOFTWARE UNIVERSITY

# Методи: type

OBJ-C

```
26  + (Car *)createBMW
27  {
28      Car *bmw = [[Car alloc] init:@"BWM" speed:300 price:5000];
29      return bmw;
30  }
```

SIGMA SOFTWARE UNIVERSITY

# Методи: мутація

Метод класу є завжди мутуючим, але лише по відношенню до властивостей.

# Індексатори

Індексатор - це метод, який дозволяє робити вибірку значення (-нь) за індексом або кількома індексами).

Swift надає наступний синтаксис опису індексатора:

```
accessLevel subscript(indexName: IndexType, …) -> ReturnType {
    // return some value(s)
}
```

# Індексатори: subscript

```swift
final public class Car {

    var modelName:String
    public var maximumSpeed:Int
    private (set) var price:Int

    var wheels:[BaseWheel] = []

    subscript(wheelIndex:Int) -> BaseWheel? {
        get {
            var wheelToReturn:BaseWheel?

            if wheelIndex < wheels.count {
                wheelToReturn = wheels[wheelIndex]
            }

            return wheelToReturn
        }

        set {
            if let wheel = newValue {
                wheels.insert(wheel, at: wheels.count)
            }
        }
    }

```

```
nil
ColorWheel
ColorWheel
```

```swift
let lastWheel = carExample[3]
print(lastWheel)
```

```
Optional(__lldb_expr_364.ColorWheel
)...
```

SIGMA SOFTWARE UNIVERSITY

# Класи

Класи

**Методи та індексатори**

Ініціалізація

Деініціалізація

Наслідування

Optional Chaining

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Класи



SIGMA SOFTWARE UNIVERSITY

# Ініціалізація

Ініціалізатор (конструктор) - це спеціальна функція із назвою *init*, метою якої є створення екземпляра типу, у якому вона описана.

Процес створення екземпляра типу з допомогою ініціалізатора називається ініціалізацією.

Звернення до ініціалізатора у межах тіла типу відбувається за назвою *init.*

Звернення до ініціалізатора поза межами тіла типу (створення екземпляра) відбувається за назвою *типу.*

# Ініціалізація

Ініціалізатор зобов’язаний *ініціалізувати* або *надати* значення за замовчуванням усім зберігаючим властивостям, які не мають такого значення на етапі їх оголошення.

Задачею ініціалізації є *підготовка* екземпляра до використання.

# Ініціалізатори: failable

Failable ініціалізатор - це ініціалізатор, який створює екземпляр Optional типу.

Swift надає два failable ініціалізатори:

```
init?() {
    // initialization or
    // return nil
}

init!() {
    // initialization or
    // return nil
}
```

SIGMA SOFTWARE UNIVERSITY

# Ініціалізація: failable

```
40      init?(speed: Int) {
41          if speed < 0 {
42              return nil
43          }
44
45          modelName = "no name"
46          maximumSpeed = speed
47          price = 0
48      }
```

SIGMA SOFTWARE UNIVERSITY

# Ініціалізація: designated vs. convenience

Ініціалізатор називається designated або призначеним, якщо він ініціалізовує усі властивості у своєму тілі.

Ініціалізатор називається convenience або допоміжним, якщо він ініціалізовує деякі властивості у своєму тілі, а решту властивостей ініціалізовує інший ініціалізатор, до якого звертається даний convenience ініціалізатор (т. зв. делегування ініціалізації).

Допоміжний ініціалізатор описується з допомогою ключового слова convenience.

# Ініціалізація: designated vs. convenience

```
50      convenience init(price:Int) {
51          self.init(name: "no name", speed: 0, price: price)
52      }
```

SIGMA SOFTWARE UNIVERSITY

# Класи



SIGMA SOFTWARE UNIVERSITY

# Класи



SIGMA SOFTWARE UNIVERSITY

# Деініціалізація

Деініціалізація - це процес, який відбувається, коли механізм ARC (automatic reference counting) або GC (garbage collection) знищує створений екземпляр та очищує виділену пам’ять з-під нього.

Процес деініціалазії відбувається автоматично, коли нема необхідності в існуванні екземпляру класу.

# Деініціалізація

Деініціалізатор (деструктор) - це спеціальна *конструкція* мови, яку Swift автоматично викликає перед тим, як пам’ять з-під екземпляра буду звільнена.

Swift надає наступний вигляд деініціалізатора:

```
deinit {
    // деініціалізація
}
```

# Деініціалізація

OBJ-C

Деініціалізатор (деструктор) - це спеціальна *конструкція* мови, яку Swift автоматично викликає перед тим, як пам’ять з-під екземпляра буду звільнена.

Objective-C надає наступний вигляд деініціалізатора:

```
 - (void)dealloc
{
    // деініціалізація
}
```

# Деініціалізація

```
66      deinit {
            print("Ah, car destroyed! :(")
68      }
```

SIGMA SOFTWARE UNIVERSITY

# Деініціалізація

```
    var bmw:Car? = Car.createBWM()
    print(bmw?.description ?? "")
    bmw = nil
92
    let lastWheel = carExample[3]
    print(lastWheel ?? "")
95
96
97
98
99
```

Car
"Car named BMW with...
nil

ColorWheel
"__lldb_expr_416.Color...

```
Car named Awesome car with speed 100 cost 1000
Car named BMW with speed 300 cost 5000
Ah, car destroyed! :(
__lldb_expr_416.ColorWheel
```

SIGMA SOFTWARE UNIVERSITY

# Деініціалізація

OBJ-C

```
26  - (void)dealloc
27  {
28      NSLog(@"Ah, car desctroyed :(");
29  }
```

SIGMA SOFTWARE UNIVERSITY

# Деініціалізація

OBJ-C

```
12  int main(int argc, const char * argv[]) {
13      @autoreleasepool {
14          Car *newCar = [[Car alloc] init:@"Awesome" speed:100 price:1000];
15          NSLog(@"%@", newCar.description);
16
17          [newCar speedUp];
18          NSLog(@"%@", newCar.description);
19
20          newCar = nil;
21      }
22      return 0;
23  }
24
25
26
27
```

```
2018-03-04 16:44:55.405892+0200 test[10128:3085009] Car named Awesome, with speed @li, cost 100
2018-03-04 16:44:55.406225+0200 test[10128:3085009] Car named Awesome, with speed @li, cost 101
2018-03-04 16:44:55.406246+0200 test[10128:3085009] Ah, car desctroyed :(
Program ended with exit code: 0
```

SIGMA SOFTWARE UNIVERSITY

# Класи



SIGMA SOFTWARE UNIVERSITY

# Класи

Класи

Методи та індексатори

Ініціалізація

Деініціалізація

**Наслідування**

Optional Chaining

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Наслідування

Наслідування - це один з принципів об’єктно-орієнтовного програмування (ООП), суть якого полягає у передачі всіх можливих властивостей, методів та індексаторів похідним класам (subclass, derived class) деякого батьківського класу (superclass, base class).

Swift представляє синтаксис наступного вигляду для реалізації механізму наслідування:

```
accessLevel class ClassName [ : ParentClassName ] {
    // declaration(s) and(or) definition(s)
}
```

Класи у Swift не наслідують жодного базового класу за замовчуванням (наприклад, NSObject).

# Наслідування

Objective-C класти в більшості наслідується принаймні від NSObject класу, або треба додати методи для створення об’єкту

```
class ClassName  : ParentClassName  {
    // declaration(s) and(or) definition(s)
}
```

# Наслідування: перевизначення та фіналізація

Похідні класи мають можливість перевизначати успадковані характеристики батьківського класу.

Swift надає синтаксис наступного вигляду для реалізації перевизначення:

```
accessLevel class Subclass: Superclass {
    override var computedProperty: ReturnType {
        return super.computedProperty
    }
    override func function(…) {
        super.function(…)
    }
    override subscript[…] -> ReturnType {
        return super[…]
    }
}
```

SIGMA SOFTWARE UNIVERSITY

# Наслідування: перевизначення та фіналізація

Батьківські класи мають можливість забороняти дочірнім класам перевизначати власні характеристики (т. зв. фіналізація характеристики).

Swift надає модифікатор final

# Наслідування: фіналізація класу

Класи можуть забороняти наслідування в цілому (т. зв. безплідні класи).

Swift надає синтаксис наступного вигляду для реалізації фіналізації класу:

```
accessLevel final class SomeClass {
    // declaration(s) and(or) definition(s)
}
```

# Наслідування та ініціалізація

Наслідування та ініціалізація визнаючають декілька основних правил:

Правило 1. Призначений ініціалізатор похідного класу зобов’язний викликати один з призначених ініціалізаторів найближчого батьківського класу.

Правило 2. Додатковий ініціалізатор деякого класу повинен викликати один з ініціалізаторів цього класу.

Правило 3. Додатковий ініціалізатор деякого класу зрештою повинен викликати один з призначених ініціалізаторів батьківського класу.

# Наслідування та ініціалізація

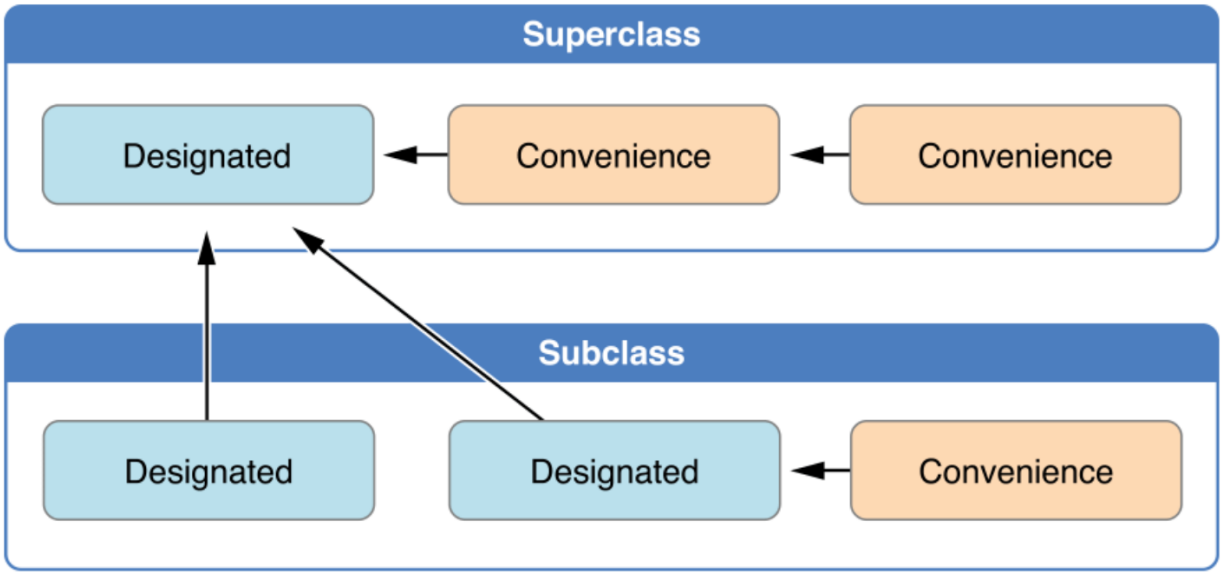
З правил випливає, що

призначені ініціалізатори завжди викликають інші призначені ініціалізатори батьківського класу (т. зв. делегація на рівні предків або “вверх”), а

додаткові ініціалізатори завжди викликають ініціалізатори того ж класу (т. зв. делегація на рівні класу)

# Наслідування та ініціалізація

Делегування ініціалізації.



# Наслідування та процес ініціалізації

Процес ініціалізації відбувається у дві фази.

Компілятор виконує чотири перевірки для того, щоб перевірити чи двофазна ініціалізація закінчилася успішно (без помилок).

# Наслідування та процес ініціалізації

Перевірки, що виконує ініціалізатор:

*Перевірка 1*. Призначений ініціалізатор повинен перевірити чи усі властивості, які описані у класі, є ініціалізованими перед делегуванням до батьківського класу.

*Перевірка 2*. Призначений ініціалізатор повинен делегувати до батьківського класу перед тим, як будуть присвоєні значення успадкованим властивостям, інакше ці властивості будуть переприсвоєні на етапі ініціалізації батьківського класу.

*Перевірка 3*. Додатковий ініціалізатор повинен делегувати до іншого ініціалізатора перед тим, як будуть присвоєні значення іншим властивостям, інакше ці властивості будуть переприсвоєні на етапі ініціалізації призначеним ініціалізатором.

*Перевірка 4*. Ініціалізатор не може звертатися до жодного методу екземпляра, властивості (читання), та у цілому звертатися через self, поки не завершиться перша фаза ініціалізації.

# Наслідування та процес ініціалізації

Перша фаза ініціалізації:

*Крок 1*. Викликається призначений або допоміжний ініціалізатор.

*Крок 2*. Виділяється пам’ять під екземпляр, однак ця пам’ять ще не ініціалізована.

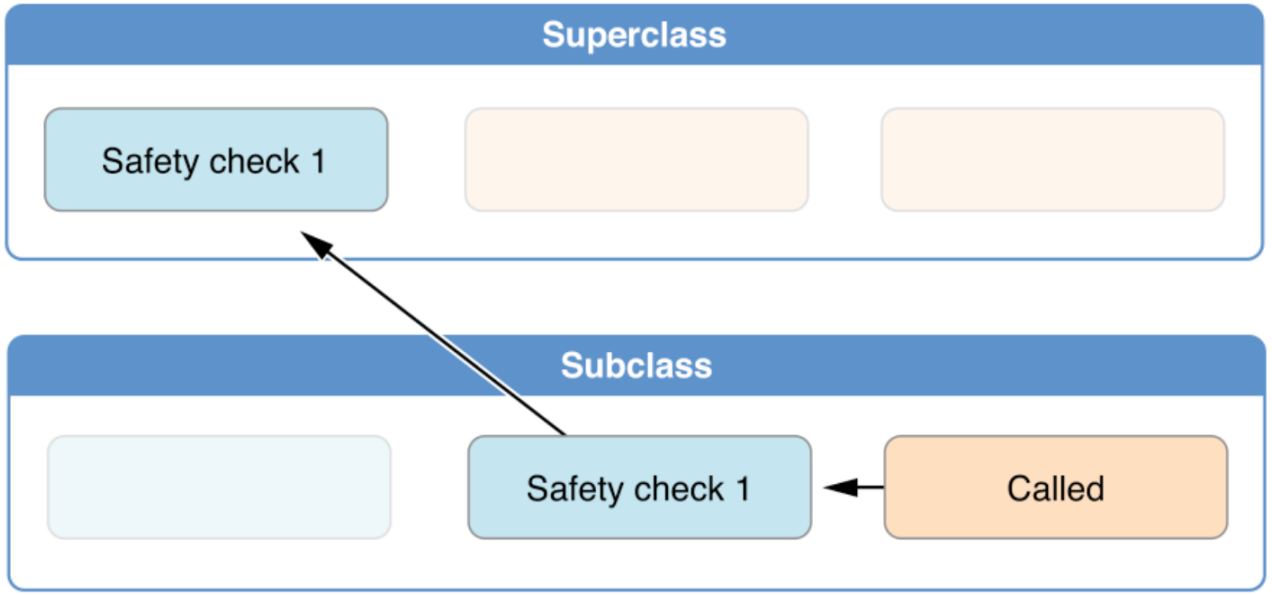
*Крок 3*. Призначений ініціалізатор класу підтверджує, що всі властивості на рівні цього класу мають присвоєні значення. Пам’ять для цих властивостей вже є ініціалізованою.

*Крок 4*. Призначений ініціалізатор делегує до призначеного ініціалізатора батьківського класу, щоб виконати дії описані вище по відношенню до властивостей батьківського класу. Процес повторюється до найвищого рівня наслідування.

*Крок 5*. Коли крок 4 закінчується і найвищий батьківський клас підтверджує, що всі його властивості ініціалізовані, то пам’ять виділена під екземпляр вважається повністю ініціалізованою, а перша фаза - завершеною.

# Наслідування та процес ініціалізації

Перша фаза ініціалізації.



# Наслідування та процес ініціалізації

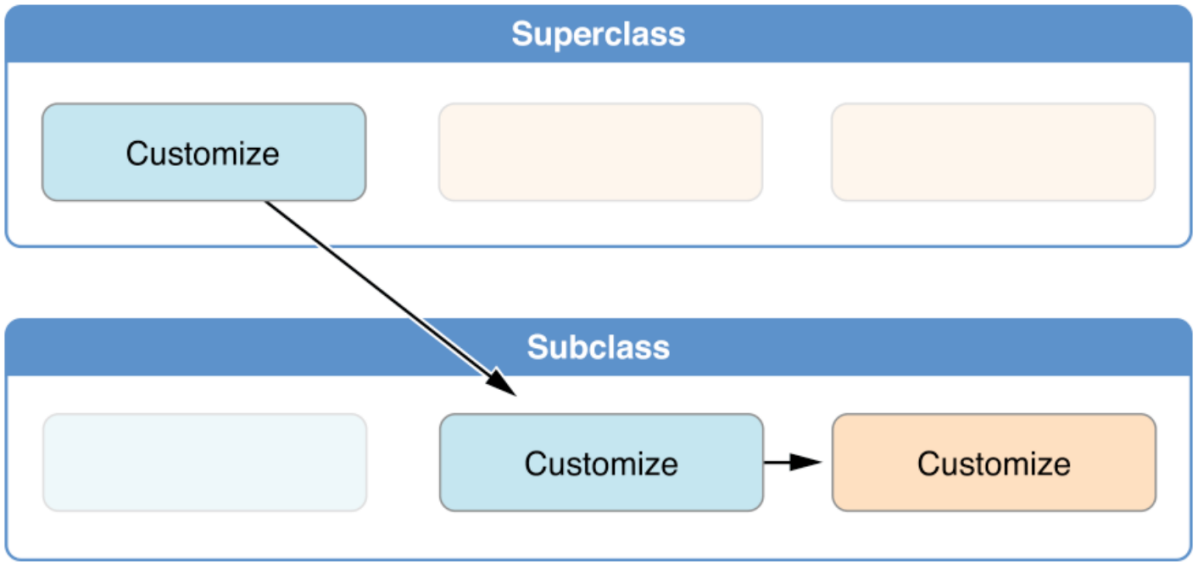
Друга фаза ініціалізації:

*Крок 1*. Починаючи з рівня найвищого батьківського класу призначений ініціалізатор може виконувати додаткові налаштування екземпляра. Звертання до self є дозволеним, так само як і до властивостей та методів екземпляру.

*Крок 2*. Накінець, кожен додатковий ініціалізатор на конкретному рівні наслідування може виконувати додаткові налаштування екземпляра. Звертання до self є дозволеним, так само як і до властивостей та методів екземпляру.

# Наслідування та процес ініціалізації

Друга фаза ініціалізації.



# Наслідування та ініціалізація

Правила автоматичного наслідування:

*Правило 1*. Якщо похідний клас не визначає жодного призначеного ініціалізатора, то він автоматично успадковує всі призначені ініціалізатори батьківського класу.

*Правило 2*. Якщо похідний клас визначає усі призначені ініціалізатори, успадковуючи їх або перевизначивши, то він автоматично успадковує усі додаткові ініціалізатори.

*Примітка*. Похідний клас може перевизначити призначений ініціалізатор батьківського класу як допоміжний.

# Наслідування та ініціалізація

Базову класи мають можливість вимагати від нащадків примусового перевизначення своїх ініціалізаторів.

Swift надає наступний синтаксис для примувого перевизначення ініціалізатора у похідних класах:

```
accessLevel class Superclass {
    required init(…) {
        // body
    }
}
```

# Класи

Класи

Методи та індексатори

Ініціалізація

Деініціалізація

**Наслідування**

Optional Chaining

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Класи

Класи

Методи та індексатори

Ініціалізація

Деініціалізація

Наслідування

**Optional Chaining**

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Optional Chaining

Optional Chaining - це процес запитів та викликів властивостей, методів, індексаторів, які можуть мати всі значення або не мати жодного

Якщо optional містить значення, то запит або виклик властивостей, методів, індексаторів є успішним.

Якщо optional не містить значення, то запит або виклик властивостей, методів, індексаторів не є успішним (для запитів повертається nil).

Декілька запитів або викликів можуть поєднуватися у ланцюжок. Якщо одна з ланок ланцюжка не проходить перевірку на успішність, то весь ланцюжок не проходить таку перевірку.

# Класи

Класи

Методи та індексатори

Ініціалізація

Деініціалізація

Наслідування

**Optional Chaining**

Класи та структури

SIGMA SOFTWARE UNIVERSITY

# Класи



SIGMA SOFTWARE UNIVERSITY

# Класи та структури

*Спільні риси класів та структур:*

Визначають властивості, методи, індексатори

Визначають ініціалізатори

Можуть бути розширені

Реалізовують протоколи

*Додаткові можливості класів (відмінності від структур):*

Визначають деініціалізатори

Підтримують наслідування

Підтримують зведення*

Типи за посилання - конртолюються ARC або GC

# Класи та структури

*Що обрати?*

Використовуйте структури, якщо сутність повинна предсталяти деякий набір характеристик, значення яких має копіюються, аніж посилатися на той самий екземпляр, при копіюванні екземпляра сутності. Структури здебільшого містять прості типи, здебільшого за значенням, а не посиланням, для яких наслідування не має змісту.

У інших випадках використовуйте класи.


SIGMA
Software
UNIVERSITY