

PG5600

iOS programming

Forelesning 3

Sist gang

- Funksjoner
- Closures
- Enumeration
- Klasser og structs
- Properties
- Metoder
- Access control

Agenda - Swift del 3

- Subscripts, Kontrutører og Arv
- deinit og ARC
- Optionals og Optional chaining
- Guard
- Type casting og Nested types
- Protocols
- Extensions
- Generics

Subscripts

- Snarveier for å hente og sette elementer i en collection, liste eller sekvens
- Sette og gette på samme måte
- Kan defineres i klasser, structs og enums

```
// Dictionary structures implementerer subscripts
```

```
var studenterIfag = ["ios": 10000, "android": 90, "wp": 10]
```

```
// Aksesser og sett elementer ved hjelp av key
```

```
print(studenterIfag["ios"]) // 10000
```

```
studenterIfag["ios"] = 500000
```

- Som kalkulerede properties, kan de være read-write eller read only

```
class EnKlassemedSubscript {  
  
    subscript (<parameters>) -> <return type> {  
        // man må ha en getter  
        get {  
            <statements>  
        }  
        // setter om man ønsker  
        set(<setter name>) {  
            <statements>  
        }  
    }  
}
```

Subscript overloading

- Definere så mange subscript man ønsker
- Type inference finner ut hvilke som skal bli brukt

```
class EnKlassemedSubscripts {  
  
    ...  
  
    subscript (pattern: String) -> Bool {  
  
    }  
  
    subscript (willBeDone: Bool) -> String {  
  
    }  
  
    ...  
}
```


Kontrutører

- Krever at man bruker navngitte parametre
- Som metoder så de omgås ved hjelp av `_`, men det anbefales ikke
- Kontanter kan settes i konstruktøren

```
class LivingThing {  
    let birth: Date  
  
    init(birth: Date) {  
        self.birth = birth  
    }  
}
```

```
var aThing = LivingThing(birth: Date())
```

- Optionals og verdier med default verdi må ikke settes i konstruktøren

```
class LivingThing {  
    let birth: Date  
    var death: Date?  
    var isAlive: Bool = true  
  
    init(birth: Date) {  
        self.birth = birth  
    }  
}  
  
var livingThing = LivingThing(birth: NSDate())
```

- Man kan ha flere konstruktører og de kan kalle hverandre
- Det finnes to forskjellige konstruktørtyper:

Designated

- Primær konstruktør som må initialisere alle ikke-optional, ikke-initialiserte properties
- Må kalle sin superclass konstruktør (ved arv)
- Det er ofte få eller bare en **Designated** konstruktør
- Alle klasser må minst ha en, med mindre man har defaultverdier på alle properties

Convenience

- Setter typisk opp en gitt state for klassen
- Krever ofte færre parametre
- Bruk de som en snarvei for å sette opp en ofte brukt state
- **Convenience** må først kalle **Designated**

```
class LivingThing {  
    let birth: Date  
    var death: Date?  
    var isAlive: Bool = true  
  
    init(birth: Date) {  
        self.birth = birth  
    }  
  
    convenience init() {  
        self.init(birth: Date())  
        self.isAlive = false // må være etter self.init  
    }  
}
```

```
var livingThing = LivingThing(birth: Date())
```

```
// convenience
```

```
var livingThing2 = LivingThing()
```

Arv

En klasse kan arve

- metoder
- properties

og alt annet fra en annen klasse

- En klasse som arver fra en annen betegnes **subclass**
- Klassen som **subclass** arver fra betegnes **superclass**
- En klasse som ikke arver av noen betegnes **base class**
- En **subclass** kan kalle metoder, properties og subscripts på **superclass**
- **subclass** kan overstyre **superclass** sine metoder, properties og subscripts

```
// base class og superclass
```

```
class LivingThing {
```

```
    let birth: Date
```

```
    var death: Date?
```

```
    // Kan ikke overskrives
```

```
    final var isAlive: Bool {
```

```
        return self.death == nil
```

```
    }
```

```
    init(birth: Date) {
```

```
        self.birth = birth
```

```
    }
```

```
    var description: String {
```

```
        return "Jeg er en levende ting som ble født \(self.birth)"
```

```
    }
```

```
}
```

```
// subclass og superclass

class Person : LivingThing {
    let firstName: String
    let lastName: String

    var fullName: String {
        return "\(self.firstName) \(self.lastName)"
    }

    // required - gjør at subclass må implementere konstruktøren
    required init(firstName: String, lastName:String, birth: NSDate) {
        self.firstName = firstName
        self.lastName = lastName
        // super kan brukes til å kalle metoder, properties og subscripts
        super.init(birth:birth)
    }

    func sayHello() -> String {
        return "Hello"
    }
}
```

```
// subclass

class Student : Person {

    // Vil gi kompile error pga required
    init {

    }

    override var description: String {
        return "Student på Westerdals med navn \$(self.fullName)"
    }

    override fun sayHello() -> String {
        return "Halla lizm"
    }

    // Compile error
    override var isAlive: Bool {
        return true
    }
}

var gunnar = Student(firstName: "Lars", lastName: "Gunnar", birth: Date())
gunnar.firstName // Lars
gunnar.description // Student på Westerdals med navn Lars Gunnar

gunnar.birth // 2014-09-07 14:17:59 +0000
```

Deinit

Deinitializer kalles rett før klassen blir fjernet fra minne

```
class Student : Person {  
  
    override var description: String {  
        return "Student på Westerdals med navn \$(self.fullName)"  
    }  
  
    override func sayHello() -> String {  
        return "Halla lizm"  
    }  
  
    deinit {  
        School.removeStudent(self.id)  
    }  
}
```

ARC

- Vanligvis håndterer ARC automatisk minne for deg, men av og til må man gjøre litt selv
- Implisitt sterk referanse
- Alt som har en referanse blir holdt i minne

```
var reference1: Student?
```

```
var reference2: Student?
```

```
reference1 = Student(firstName: "Lars", lastName: "Gunnar", birth: NSDate()) // sterk referanse
```

```
reference2 = reference1 // To sterke referanser til Lars
```

```
reference1 = nil // en sterk referanse igjen
```

```
reference2 = nil // ingen referanse igjen, instansen blir fjernet fra minne og deinit blir kalt
```


- Hvordan løser vi sirkulære avhengigheter?
- Bruk **weak** for å si at man ikke ønsker å øke referansetelleren
- Man kan ikke bruke **weak** på kontanter, da en weak vil kunne endre seg runtime

```
class Student : Person {  
    ...  
  
    weak var school: School?  
  
    ...  
}  
  
class School {  
  
    ....  
  
    var students = [Student]()  
  
    ....  
}
```

- Hvis referansen kan bli nil en gang i løpet av applikasjonens kjøretid, bruk **weak**
- Bruk **unowned** i stedet for weak der du vet at verdien alltid vil være satt

```
class Student : Person {  
    ...  
  
    unowned var school: School?  
  
    ...  
}
```

```
class School {  
  
    ....  
  
    var students = [Student]()  
  
    ....  
}
```

**Se boken for mer
informasjon om
minnehåndtering**

Under kapitlet -> Automatic Reference Counting

Optional Chaining

```
if let street = westerdals.students.first?.address?.street {  
    print("Studenten bor i \$(street).")  
} else {  
    print("Kunne ikke hente gatenavn")  
}
```

- Du kan akserssere properties
- Kalle metoder
- Kalle subscript

Type Casting

is

- brukes til å sjekke typen til en instans

as

- brukes til å behandle en instans som om det var en annen type i dens typetre

```
class LivingThing {}  
class Person: LivingThing {}  
class Animal: LivingThing {}
```

```
let living = [  
    Person(birth: NSDate()),  
    Animal(birth: NSDate()),  
    Person(birth: NSDate()),  
    Animal(birth: NSDate()),  
    Animal(birth: NSDate())  
]
```

```
living[0] is Person // true  
living[1] is Animal // true  
living[2] is Animal // false
```

as?

```
for item in living {  
    if let person = item as? Person {  
        print("Is alive: \$(person.isAlive)")  
    } else if let animal = item as? Animal {  
        print("\$(animal.roar())")  
    }  
}
```

Any og AnyObject

- AnyObject kan representere en instans hvilke som helst klassetype
- Any kan representere en instans av hvilke som helst type, inkludert funksjontyper
- Bør bare brukes når man faktisk trenger det, vær eksplisitt

```
// Cocoa-APIer og array vil innholde Any
```

```
let someObjects: [AnyObject] = [  
    Person(birth: NSDate()),  
    Person(birth: NSDate()),  
    Person(birth: NSDate())  
]  
  
for person in someObjects as [Person] {  
    print("Is alive: \(person.isAlive)")  
}
```

```
var things = [Any]()

things.append(0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))

for thing in things {
    switch thing {
    case 0 as Int:
        print("Det var en int som var 0 der ja")
    case let someInt as Int:
        print("Fant en int som er \$(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("en positiv Double \$(someDouble)")
    case is Double:
        print("En eller annen Double var også der gitt")
    case let someString as String:
        print("fant en string som inneholder \"\$(someString)\"")
    case let (x, y) as (Double, Double):
        print("en (x, y) verdi der x \$(x), y \$(y)")
    default:
        print("noe annet greier")
    }
}
```

Guard

```
let rocketDictionary : [String : String]? = [ "name" : "Falcon 9", "fuelName" : "liquid oxygen" ]
```

```
var factoryRobotsReady = true
```

```
func generateRocket(rocketDictionary: [String : String]?) {  
    if factoryRobotsReady {  
        if let actualDictionary = rocketDictionary {  
            if let rocketName = actualDictionary["name"] {  
                if let fuelName = actualDictionary["fuelName"] {  
                    Rocket(rocketName, fuelName: fuelName)  
                }  
            }  
        }  
    }  
}
```


Guard

```
func generateRocketSchematics(rocketDictionary : [String : String]?) {

    guard factoryRobotsReady else {
        print("robots not ready")
        return
    }

    guard let actualDictionary = rocketDictionary else {
        print("no data to generate schematics")
        return
    }

    guard let rocketName = actualDictionary["name"] else {
        print("no rocket name")
        return
    }

    guard let fuelName = actualDictionary["fuelName"] else {
        print("no fuel name")
        return
    }

    Rocket(rocketName, fuelName: fuelName)
}
```

Extentions

- Utvide funksjonalitet for en bestemt type
- Vanlig og static kalkulererte properties
- Definere nye instansmetoder og klassemetoder
- Nye init metoder
- Nye subscripts
- Definere ny nestet type
- Gir mulighet å implementere en protocol for en eksiterende type

```
extension String {  
    var uppercase: String { return self.toUpperCaseString }  
}
```

```
var name = "Hans Magnus"  
name.toUpperCase // "HANS MAGNUS"
```

Protocols

- *Samme som interface i Java og andre språk*
- *Definerer opp et sett med metoder, properties, klasse metoder, operatorer og subscripts som passer en bestemt funksjonalitet*
- *Inneholder ingen implementasjonskode*

```
protocol LivingThing {  
    var mustBeSettable: Int { get set }  
    var doesNotNeedToBeSettable: Int { get }  
  
    class func someTypeMethod()  
    func random() -> Double  
    mutating func toggle() // gjør det mulig å endre properties  
}
```

- En protocol kan brukes alle steder hvilke som helst type ville bli brukt
- En klasse, struct eller enum kan implementere flere protocols
- Protocols kan arve av hverandre
- Mer om protocols når vi går over til iOS

Protocol extensions

- Du kan lage implementasjonskode for metoder
- Apple introduserte Protocol oriented development

```
extension LivingThing {  
  
    func random() -> Double {  
        return 42  
    }  
}
```

Generics

- Mye av Swift sitt standard bibliotek er bygd med generics kode
- Eksempelvis er Array og Dictionary av typene generic collections
- Kan definere at typen i det minste skal implementere en protocol "Type Constraints"

Generic functions

```
func printSequence<T: SequenceType>(sequence: T) {  
    for part in sequence {  
        print(part)  
    }  
}
```

```
printSequence("ABCDEF")
```

```
printSequence(["Aa", "Bb"])
```

```
printSequence(["A": "B", "B": "A"])
```

Generic Types

- Enums, structs og klasser kan også være generiske
- Array og Dictionary er eksempler på generiske structs

```
class GenericClass<T> {  
    var object: T  
  
    init(object: T) {  
        self.object = object  
    }  
  
    func getObject() -> T {  
        return self.object;  
    }  
  
    func prinObject() {  
        print("Type of T is \(self.object)");  
    }  
}  
  
var a = GenericClass<Int>(object: 1)  
a.prinObject()
```

Associated Types

- I en protokoll kan man lage et alias (associated type) der det er opp til implementasjonen å definere den faktiske typen.
- Dette er for å kunne referere til typen i metoder og subscripts uten at man bestemmer typen i protokollen

```
protocol Container {  
    typealias ItemType  
    mutating func append(item: ItemType)  
    var count: Int { get }  
    subscript(i: Int) -> ItemType { get }  
}
```

```
class Example: Container {  
    typealias ItemType = String  
    var array = [ItemType]()  
  
    func append(item: ItemType) {  
        self.array.append(item)  
    }  
  
    var count: Int {  
        get {  
            return array.count  
        }  
    }  
  
    subscript(i: Int) -> ItemType {  
        get {  
            return array[i]  
        }  
    }  
}
```

Operator overloading

```
func +(left: Balloon, right: Balloon) -> [Balloon] {  
    return [left, right]  
}
```

```
let balloon1 = Balloon()
```

```
let balloon2 = Balloon()
```

```
let array = balloon1 + balloon2
```

Videre lesning

- Se TSPL-boka om hovedtema i forelesningen

Oppgaver

- Se IT's learning

Forelesningen er basert på fjorårets foiler, laget av

Hans Magnus Inderberg og Mads Mobæk