

# PG5600

## IOS PROGRAMMING

### FORELESNING 2

**HVA SKJEDDE PÅ TIRSDAG?**

# IPHONE 6

4.7 inches  
iPhone 6



5.5 inches  
iPhone 6 Plus



# APPLE WATCH



TUE 9 10:09

12:00PM-1:30PM  
Lunch with Ben  
Workshop Cafe

72°  LON  
6:09

# XCODE 6 GM SEED

```
5 // Created by Hans M. Inderburg on 9/10/14.
6 // Copyright (c) 2014 HM&M. All rights reserved.
7 //
8
9
10
11
12
13
14 var window: UIWindow?
15
16
17 func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
18     // Override point for customization after application launch.
19     return true
20 }
21
22 func applicationWillResignActive(application: UIApplication) {
23     // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interruptions
24     // (such as an incoming phone call or SMS message) or when the user quits the application and it begins the transition to the
25     // background state.
26     // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games should use this method to
27     // pause the game.
28 }
29
30 func applicationDidEnterBackground(application: UIApplication) {
31     // Use this method to release shared resources, save user data, invalidate timers, and store enough application state information to
32     // restore your application to its current state in case it is terminated later.
33     // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.
34 }
35
36 func applicationWillEnterForeground(application: UIApplication) {
37     // Called as part of the transition from the background to the inactive state; here you can undo many of the changes made on entering
38     // the background.
39 }
40
41 func applicationDidBecomeActive(application: UIApplication) {
42     // Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously in the
43     // background, optionally refresh the user interface.
44 }
45
46 func applicationWillTerminate(application: UIApplication) {
47     // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:
48 }
```

Text Setting

Text Encod

Line End

Indent U

W

Source Cor

Reposi

T

Current Bra

Ver

St

Loca





# IOS 8 GM SEED



The biggest iOS  
release ever.

Coming Soon



# 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
- » 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 structs implementerer subscripts
```

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

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

```
println(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: NSDate  
  
    init(birth: NSDate) {  
        self.birth = birth  
    }  
}
```

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



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

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

- » Man kan ha flere kontruktører og de kan kalle hverandre
- » Det finnes to forskjellige kontruktørtyper:

## Designated

- » Primær konstruktør som må initialisere alle properties
- » Må kalle sin superclass konstruktør
- » Det er ofte få eller bare en Designated konstruktør
- » Alle klasser må minst ha en

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

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

## 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 til slutt kalle Designated

```
class LivingThing {
    let birth: NSDate
    var death: NSDate?
    var isAlive: Bool = true

    init(birth: NSDate) {
        self.birth = birth
    }

    convenience init() {
        self.init(birth: NSDate())
        self.isAlive = false // må være etter self.init
    }
}

var livingThing = LivingThing(birth: NSDate())

// convenience
var livingThing2 = LivingThing()
```

**ARW**



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: NSDate  
    var death: NSDate?  
  
    // Kan ikke overskrives  
    final var isAlive: Bool {  
        return self.death == nil  
    }  
  
    init(birth: NSDate) {  
        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: NSDate())
```

```
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)  
    }  
}
```

**ARE**



- » 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 MINNEHÅNTERING

UNDER KAPITELLET -> AUTOMATIC REFERENCE COUNTING

# OPTIONAL CHAINING



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

» Du kan akseressere 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 {  
        println("Is alive: \[person.isAlive]")  
    } else if let animal = item as? Animal {  
        println("\[animal.roar()]")  
    }  
}
```

**ANY OG ANYOBJECT**

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

```
// Cocoa apis og array vil alltid inneholde AnyObject,  
// da Objective-C ikke har eksplisitte typede arrays
```

```
let someObjects: [AnyObject] = [  
    Person(birth: NSDate()),  
    Person(birth: NSDate()),  
    Person(birth: NSDate())  
]  
  
for person in someObjects as [Person] {  
    println("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:
        println("Det var en int som var 0 der ja")
    case let someInt as Int:
        println("Fant en int som er \${someInt}")
    case let someDouble as Double where someDouble > 0:
        println("en positiv Double \${someDouble}")
    case is Double:
        println("En eller annen Double var også der gitt")
    case let someString as String:
        println("fant en string som inneholder \"\${someString}\"")
    case let (x, y) as (Double, Double):
        println("en (x, y) verdi der x \${x}, y \${y}")
    default:
        println("noe annet greier")
    }
}
```

# NESTED TYPES

» Man kan ha klasser, structs og enums nestet i hverandre

```
struct Student {  
    enum Mood: String {  
        case Sad = ":(", Happy = ":)"  
    }  
}
```

```
Student.Mood.Sad.toRaw()
```

**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 eksisterende type

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

```
var name = "Hans Magnus"  
name.uppercase // "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



**GENERIC**

- » 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 {  
        println(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() {  
        println("Type of T is \(${self.object})");  
    }  
}  
  
var a = GenericClass<Int>(object: 1)  
a.prinObject()
```

# ASSOCIATED TYPES

» I en protocol 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 append og subscript

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

# WHERE

```
func allItemsMatch<
  C1: Container, C2: Container
  where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>

  (someContainer: C1, anotherContainer: C2) -> Bool
{
  // funksjonskropp
}
```

**OPPGAVER**  
**SE ØVINGSOPPGAVERNE**