13 Standardbibliotek (provavsnitt)

13.1 Standardbibliotek
13.2 Matematiska konstanter och funktioner
13.3 Typerna Any och AnyObject
13.4 Textrepresentation av värden
13.5 Likhetsjämförelse av värden
13.7 Storleksjämförelse av värden
13.8 Bitvektorer
13.9 Kommandoradsargument

Standardbibliotek

Programspråket Swift har tillgång till flera standardbibliotek:

- Swifts eget standardbibliotek
  - ➜ Grundläggande datatyper, t ex `Int`, `Double`, `Bool`, `Character` och `String`
  - ➜ Typer för datastrukturer, t ex `Array`, `Set` och `Dictionary`
  - ➜ Grundläggande gränssnitt, t ex `CustomStringConvertible`, `Error` och `Hashable`
  - ➜ Globala funktioner, t ex `assert`, `max`, `min`, `print` och `readLine`
  - ➜ Integration med programspråken **C** och **Objective-C**
- **Foundation**-ramverket – delas av Swift och Objective-C
  - ➜ Wrapperobjekt av klasser som t ex `NSNumber`, `NSValue`, `NSData` och `NSMutableData`
  - ➜ Texthantering via klasserna `NSString` och `NSMutableString`
  - ➜ Datastrukturer av klasser som t ex `NSArray`, `NSSet` och `NSDictionary`
  - ➜ Hantering av datum och tid via klasserna `NSDate`, `NSCalendar` och `NSDateFormatter`
  - ➜ Hantering av in- och utmatning, t ex formaterad utmatning och filhantering
  - ➜ Stöd för flertrådad exekvering
- Därutöver tillkommer olika bibliotek beroende på målplattform, t ex
  - ➜ **Cocoa** vid utveckling av applikationer för **macOS**
  - ➜ **Cocoa Touch** vid utveckling av applikationer för **iOS**

Programspråket Swift har tillgång till flera olika standardbibliotek:

- Swift har ett eget standardbibliotek som definierar de grundläggande datatyperna och deras operatorer, typer för datastrukturer (framför allt `Array`, `Set` och `Dictionary`), en lång rad grundläggande gränssnitt och en uppsättning globala funktioner. Dessutom ingår särskilt stöd för integrationen med programspråken **C** och **Objective-C**, integration med plattformen (t ex kommandoradsargument) samt integration med playground-filer.
- Swift har även tillgång till **Foundation**-ramverket, som språket delar med Objective-C. Foundation-ramverket har en genomgående objektorienterad design och består av klasser för bl a wrapperobjekt (som kapslar in värden som inte är objekt), texthantering (bl a `NSString` och `NSMutableString`), datastrukturer (`NSArray`, `NSSet` och `NSDictionary` samt deras ändringsbara kusiner), hantering av datum och tid (`NSDate`, `NSLocale`, `NSTimeZone`, `NSCalendar` och `NSDateFormatter`), inmatning och utmatning (t ex formaterad utmatning och filhantering) samt stöd för flertrådad exekvering. I version 3 av Swifts standardbibliotek finns åtskilliga typer som fungerar som en brygga till dessa klasser, t ex `Date`, `Locale` och `Calendar`. För att använda Foundation-specifika klasser krävs att källkodsfilen innehåller en `import`-sats:

  ```
  import Foundation
  ```

- Utöver dessa bibliotek som Swift alltid har tillgång till finns stora tilläggsbibliotek beroende på vilken målplattform utveckling sker för, t ex **Cocoa** vid utveckling av applikationer för **macOS**, **Cocoa Touch** vid utveckling av applikationer för **iOS**, etc. Dessa bibliotek är omfattande och består i sin tur av många ramverk.

Programspråket C:s standardbibliotek

Matematiska konstanter och funktioner

Matematiska konstanter och funktioner är tillgängliga via C:s standardbibliotek.

Rätt bibliotek måste importeras:

- Använd `import Darwin` för Apples plattformar

  (importeras automatiskt av bl a **Foundation**, **UIKit** och **AppKit**)
- Använd `import Glibc` för Linux

```
import Darwin

let radius = 7.5                    // Cirkelns radie
let area = M_PI * radius * radius   // Cirkelns area

let sq = sqrt(radius)               // Kvadratroten av radien
let r3 = pow(radius, 3.0)           // Radien upphöjt till 3
let sqRounded = lround(sq)          // Avrunda till närmaste heltal

let randomNumber = Double(arc4random()) / Double(UInt32.max)
                                    // Slumptal mellan 0 och 1
let diceRoll = Int(arc4random_uniform(6) + 1)
                                    // Slumpmässigt tärningsslag
```

Matematiska konstanter och funktioner är tillgängliga i Swift via programspråket C:s standardbibliotek (precis som i Objective-C). För att få tillgång till C:s standardbibliotek måste rätt bibliotek importeras:

- På Apples plattformar importeras C:s standardbibliotek med satsen `import Darwin`.

  Detta är emellertid inte nödvändigt om koden redan importerar ett annat bibliotek som i sin tur importerar Darwin, t ex **Foundation**, **UIKit** eller **AppKit**.
- På plattformen **Linux** importeras C:s standardbibliotek med satsen `import Glibc`.

I biblioteket finns många matematiska konstanter och funktioner:

- Konstanten π är tillgängliga via den globala konstanten `M_PI`.
- Matematiska funktioner som t ex kvadratrot och potens är tillgängliga i form av globala funktioner, t ex `sqrt(_:)` och `pow(_:_:)`. Observera att Swifts starka typkontroll kräver att vi använder argument av rätt typ – i dessa fall `Double`.
- Det finns flera globala funktioner för avrundning av flyttal. Funktionen `lround(_:)` avrundar ett flyttal av typen `Double` till närmaste heltal av typen `Int`. Funktionen `round(_:)` utför samma avrundning, men lämnar svaret som ett flyttal av typen `Double`.
- Slumptal kan genereras med den globala funktionen `arc4random()`, som genererar likformigt fördelade slumpmässiga heltal i intervallet fr o m `0` t o m `UInt32.max`. Den globala funktionen `arc4random_uniform(_:)` genererar likformigt fördelade slumpmässiga heltal i intervallet fr o m `0` t o m argumentet minus `1`.

Swifts standardbibliotek

Typerna Any och AnyObject

Swifts standardbibliotek definierar två generella typer:

- Typen Any
  - ➜ Representerar en helt godtycklig typ i Swift
  - ➜ Omfattar såväl grundläggande datatyper, uppräknade datatyper, posttyper som klasser
  - ➜ Kan t ex användas som argument- eller returtyp för ett värde av en godtycklig typ
- Typen `AnyObject`
  - ➜ Representerar en referens till ett objekt av en godtycklig klass
  - ➜ Motsvarar typen `id` i **Objective-C**
  - ➜ Förekommer ofta i kommunikationen med **Cocoa**- och **Cocoa Touch**-biblioteken
  - ➜ Används ofta i biblioteken för datastrukturer av objekt utan känd typ

Foundation-ramverket utnyttjar en gemensam moderklass, `NSObject`.

Klassen `NSObject` är inte samma som typen `AnyObject`!

Swifts standardbibliotek definierar två generella typer som kan användas för att representera värden av olika typer:

- Typen `Any` representerar värden av en helt godtyckligt typ i Swift. Det innebär att värdet kan vara av en grundläggande datatyp, en datastrukturtyp, en uppräknad datatyp, en posttyp, en klass eller en funktionstyp. Typen `Any` förekommer bl a som argument- eller returtyp i funktioner och metoder som ska kunna hantera värden av en helt godtycklig typ.
- Typen `AnyObject` representerar en referens till ett objekt av en godtycklig klass i Swift. Detta är Swifts motsvarighet till typen `id` i programspråket **Objective-C**. Typen `AnyObject` förekommer ofta i deklarationer av typer för egenskaper, argument eller returvärden i **Cocoa**- och **Cocoa Touch**-biblioteken, i synnerhet som grundtyp för element i datastrukturer där objektens exakta klasstillhörighet inte är känd.

Foundation-ramverket definierar en klass `NSObject`, som är en moderklass (gemensam högsta överklass) för alla klasser i Foundation. Observera att `NSObject` inte är samma som `AnyObject`!

Swifts standardbibliotek

Textrepresentation av värden

```
class Account : CustomStringConvertible, Exportable {
    let accountNo: String
    private(set) var balance: Double

    var description: String {
        return "Account[accountNo=\(accountNo), balance=\(balance)]"
    }

    init(no: String, balance: Double) throws {
        self.accountNo = no
        self.balance = balance
    }

    ...
}
```

```
let account1 = try! Account(no: "PS721-39", balance: 15000)
print("account1 = \(account1)")
```

- Egenskapen `description` definierar en textrepresentation av ett värde
- Används bl a av initieringsuttrycket `String(value)`, t ex i `print(_:)`

Om vi skriver ut ett värde av en av Swifts fördefinierade datatyper via biblioteksfunktionen `print(_:)` eller bäddar in ett värde i en textsträng via uttryck av typen `\(value)` får vi en tydlig utskrift av värdet:

```
let answer = 42
print("Svaret är \(answer).")         // "Svaret är 42."
```

För en egendefinierad typ som klassen `Account` ger ett motsvarande försök ett betydligt mindre upplysande resultat:

```
let account1 = try! Account(no: "PS721-39", balance: 15000)
print(account1)                       // "Account"
```

Vi kan påverka hur ett värde av en egendefinierad typ representeras som text genom att låta typen implementera gränssnittet `CustomStringConvertible` från Swifts standardbibliotek:

```
class Account : CustomStringConvertible, Exportable { … }
```

För att implementera gränssnittet `CustomStringConvertible` ska typen implementera en egenskap av typen `String` med namnet `description`. Egenskapen behöver endast ges läsrättigheter. Det normala sättet att göra detta är via en beräknad egenskap:

```
var description: String {
    return "Account[accountNo=\(accountNo), balance=\(balance)]"
}
```

Utskriften med biblioteksfunktionen `print(_:)` ger nu ett betydligt mera hjälpsamt resultat:

```
print(account1)      // "Account[accountNo=PS721-39, balance=15000.0]"
```

Swifts standardbibliotek

Likhetsjämförelse av värden

```
class Account : CustomStringConvertible, Hashable, Exportable {
    let accountNo: String
    private(set) var balance: Double

    var hashValue: Int {
        return accountNo.hashValue
    }

    ...
}

func ==(lhs: Account, rhs: Account) -> Bool {
    return lhs.accountNo == rhs.accountNo
}
```

```
let account1 = try! Account(no: "PS721-39", balance: 15000)
let account2 = try! Account(no: "PS721-39", balance: 0)
if account1 == account2 {
    print("Samma konto!")
}
```

- Om vi definierar operatorn == får vi automatiskt tillgång till operatorn `!=`

För Swifts fördefinierade datatyper, inklusive typerna för datastrukturer, är likhetsjämförelse med operatorerna `==` och `!=` definierad. Dessutom är operatorerna automatiskt definierade för typer av sammansatta värden bestående av högst sex värden samt för uppräknade datatyper utan anknutna värden.

För övriga typer är operatorerna `==` och `!=` inte definierade. Det leder t ex till kompileringsfel att försöka jämföra två objekt av klassen `Account` med någon av operatorerna `==` och `!=`. Vi kan ändra detta genom att själva definiera operatorerna med ett lämpligt beteende, där vi bestämmer vilka objekt som räknas som "lika".

För att definiera likhetsjämförelse för en egendefinierad typ krävs att vi implementerar gränssnittet `Equatable`, som stipulerar ett krav: operatorn `==` ska definieras som en global funktion med namnet `==`. Funktionen ska ha två argument av den egendefinierade typen och returtypen `Bool`:

```
func ==(lhs: Account, rhs: Account) -> Bool { … }
```

Funktionen ska returnera `true` om de båda värdena anses lika, annars `false`. Hur jämförelsen utförs beror helt på typens innebörd. För klasser av objekt är det lämpligt att bygga jämförelsen på data som inte förändras under objektens livstid. För bankkonton förefaller det vara rimligt att anse att två bankkonton är lika om de har samma kontonummer. Jämförelsen är lätt att utföra, eftersom typen `String` stödjer jämförelse med operatorn `==`:

```
func ==(lhs: Account, rhs: Account) -> Bool {
    return lhs.accountNo == rhs.accountNo
}
```

Det går nu att jämföra objekt av klassen `Account`:

```
if account1 == account2 {
    print("Samma konto!")
}
```

På köpet får automatsikt vi tillgång till operatorn `!=`, som jämför om två värden är olika:

```
if account1 != account2 {
    print("Olika konton!")
}
```

Fr o m Swift 3.0 är det alternativt tillåtet att definiera operatorn som en typmetod i stället för en global funktion. Den ska då deklareras antingen med `static` eller `class final`:

```
class final func ==(lhs: Account, rhs: Account) -> Bool {
    return lhs.accountNo == rhs.accountNo
}
```

I många fall är det fördelaktigt att inte enbart implementera gränssnittet `Equatable`, utan i stället implementera gränssnittet `Hashable`, som är ett undergränssnitt till `Equatable`:

```
class Account : CustomStringConvertible, Hashable …
```

Gränssnittet `Hashable` tillför ytterligare ett krav: att implementera en egenskap av typen `Int` med namnet `hashValue`. Egenskapen behöver endast definieras med läsrättigheter. Det normala sättet att göra detta är via en beräknad egenskap.

Metoden `hashValue` används bakom draperierna i Swift för att generera hashkoder som bl a används för effektiv indexering i datastrukturer. En *hashkod* är ett heltal som för varje värde av en typ valts på sådant sätt att samma värde alltid har samma hashkod under en exekvering av programmet (medan det är önskvärt att olika värden ska ha en god chans att få olika hashkoder). Element som ingår i mängder av typen `Set` och nycklar i associativa datastrukturer av typen `Dictionary` måste implementera gränssnittet `Hashable`.

Specifikationen kräver att metoden `hashValue` måste returnera samma heltal för två värden av en typ om operatorn `==` bedömt dem som lika. Vi behöver därför skriva en egen metod för `hashValue` som stämmer överens med vår implementation av operatorn `==`.

De grundläggande datatyperna inklusive typen `String` samt uppräknade datatyper utan anknutna värden implementerar redan gränssnittet `Hashable`. Vi kan därför implementera metoden `hashValue` för klassen `Account` enkelt genom att definiera hashkoden för ett bankkonto som kontonumrets hashkod:

```
var hashValue: Int {
    return accountNo.hashValue
}
```

Swifts standardbibliotek

Storleksjämförelse av värden

```
struct ISODate : Comparable {
    var year:  Int
    var month: Month
    var day:   Int
    var dayNumber: Int { ... }

    ...
}

func ==(lhs: ISODate, rhs: ISODate) -> Bool {
    return lhs.year == rhs.year && lhs.dayNumber == rhs.dayNumber
}

func <(lhs: ISODate, rhs: ISODate) -> Bool {
    if lhs.year == rhs.year {
        return lhs.dayNumber < rhs.dayNumber
    } else {
        return lhs.year < rhs.year
    }
}
```

- Vi behöver inte själva definiera operatorerna !=, <=, > och >=

På samma sätt är storleksjämförelse med operatorerna <, <=, > och >= endast fördefinierad för vissa av Swifts typer, men även här kan vi definiera dem för en egen typ. T ex skulle posttypen ISO`Date`, som vi presenterade i kapitlet *Poster och egenskaper*, kunna ge stöd för kronologisk jämförelse av datum.

Storleksjämförelse av värden av samma typ definieras av gränssnittet `Comparable` i Swifts standardbibliotek, så det första steget är att posttypen `ISODate` förbinder sig att implementera `Comparable`:

```
struct ISODate : Comparable { … }
```

`Comparable` är ett undergränssnitt till gränssnittet `Equatable` i Swifts standardbibliotek. Därmed är vi bundna att även uppfylla detta gränssnitt, som stipulerar att det ska finnas stöd för operatorn ==. Vi löser detta på samma sätt som i föregående exempel. I implementationen drar vi nytta av den beräknade egenskapen `dayNumber`, som returnerar ett löpande dagnummer inom ett år:

```
func ==(lhs: ISODate, rhs: ISODate) -> Bool {
    return lhs.year == rhs.year && lhs.dayNumber == rhs.dayNumber
}
```

Det krav som tillkommer i gränssnittet `Comparable` är att vi även ska definiera operatorn <, som ska avgöra om den vänstra operanden är "mindre än" (dvs i vårt fall kronologiskt ordnad före) den högra operanden. Vi gör detta via ytterligare en global funktion med namnet <. Funktionen ska ha två argument av den egendefinierade typen och returtypen `Bool`:

```
func <(lhs: ISODate, rhs: ISODate) -> Bool { … }
```

Det går nu att jämföra värden av posttypen `ISODate` med samtliga sex jämförelseoperatorer. De fyra återstående operatorerna får vi automatiskt från standardbiblioteket när väl == och < är definierade.

Swifts standardbibliotek

Bitvektorer

Uppsättningar av alternativ hanteras i biblioteken ofta som *bitvektorer*.

- Om ett alternativ ingår i uppsättningen sätts en unik bit till värdet `1`.
- I Swift 1 måste bitvisa operatorer användas för att sätta och testa alternativ.

Detta förenklas numera av gränssnittet `OptionSet`:

```
struct Directions : OptionSet {
    let rawValue: Int
    static let south = Directions(rawValue: 1)
    static let west  = Directions(rawValue: 2)
    static let north = Directions(rawValue: 4)
    static let east  = Directions(rawValue: 8)
}

let exits: Directions = [.west, .north]
var triedExits: Directions = []
if exits.contains(.north) {
    triedExits.insert(.north)
}
```

- Via gränssnittet kan vi hantera bitvektorerna som om de var mängder.

Ett vanligt behov är att hantera en uppsättning av alternativ, där noll, ett eller flera alternativ kan förekomma samtidigt. I Apples bibliotek hanteras sådana uppsättningar av alternativ ofta som bitvektorer. En *bitvektor* är ett heltal där varje alternativ tilldelas en egen unik bit som kan vara antingen `0` eller `1`. I version 1 av Swift hanterades bitvektorer via de bitvisa operatorer som ingår i språket. Via dessa kan enskilda bitar sättas, nollställas och testas. Se kapitlet *Grundläggande språkelement* för mer detaljer!

Att hantera bitvektorer med hjälp av de bitvisa operatorerna kan lätt leda till fel. Numera finns det ett mycket enklare sätt. Genom att låta bitvektorns typ implementera gränssnittet `OptionSet` från Swifts standardbibliotek (`OptionSetType` i Swift 2) kan vi hantera bitvektorn som om den vore en mängd.

I exemplet definieras en posttyp `Directions`, som ska implementera en bitvektor där uppsättningen av alternativ är de olika väderstrecken: söder, väster, norr och öster. En sådan bitvektor skulle t ex kunna representera möjliga utgångar från ett rum i ett spel. Vi låter posttypen implementera gränssnittet `OptionSet` och definierar typkonstanter för de enskilda väderstrecken. Varje typkonstant sätts till värdet av den unika bit som representerar just det väderstrecket:

```
struct Directions : OptionSet {
    let rawValue: Int
    static let south = Directions(rawValue: 1)
    ...
}
```

Genom att typen implementerar gränssnittet `OptionSet` får vi tillgång till en syntax som liknar den för mängder. Värden kan initieras från vektorer, och via en implementationsutbyggnad finns det redan metoder som t ex `insert(_:)` (lägga till ett alternativ i uppsättningen) och `contains(_:)` (testa om ett alternativ ingår i uppsättningen).

Swifts standardbibliotek

Kommandoradsargument

Via `CommandLine.arguments` kan vi komma åt kommandoradsargument:

main.swift

```
let args = CommandLine.arguments
print("Exekverande program: \(args[0])")
for i in 1..<args.count {
    print("Argument \(i): \(args[i])")
}
```

- `CommandLine.arguments` är en vektor av textsträngar
- Vektorns första element (position `0`) är programmets namn
- Samtliga efterföljande element i vektorn är kommandoradsargumenten

Foundation-klassen `ProcessInfo` kan ge mer detaljerad information.

Program utvecklade i Swift kan exekveras från kommandoraden. Värdet `arguments` hos den uppräknade datatypen `CommandLine` i Swifts standardbibliotek (`Process` före Swift 3.0) ger oss tillgång till en vektor av textsträngar som innehåller de kommandoradsargument som kan ha ingått i det kommando som startade programmet:

- Vektorns första argument (dvs position `0` i vektorn) är programmets namn, dvs kommandonamnet i det kommando som startade programmet:

```
let args = CommandLine.arguments
print("Exekverande program: \(args[0])")
```

- Samtliga efterföljande element i vektorn är kommandoradsargumenten, där det första argumentet har positionsnummer `1` i vektorn:

```
for i in 1..<args.count {
    print(""Argument \(i): \(args[i]))
}
```

Klassen `ProcessInfo` i **Foundation**-ramverket (`NSProcessInfo` före Swift 3.0) innehåller mer detaljerad information om det exekverande programmet.