13 Standardbibliotek (provavsnitt)

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`
  - ➜ Grafiska användargränssnitt för samtliga Apples plattformar via ramverket **SwiftUI**
- **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 posttyperna `Date`, `Locale`, `Calendar` och `TimeZone`
  - ➜ 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. Fr o m **Swift 5.1** innehåller Swifts standardbibliotek dessutom ramverket **SwiftUI**, som kan användas för att utveckla användargränssnitt för samtliga Apples plattformar.
- 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. Numera innehåller Foundation-ramverket även posttyper som fungerar som en brygga till dessa klasser, t ex `Date`, `Locale` och `Calendar`. För att använda Foundation-specifika typer 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 Touch** vid utveckling av applikationer för **iOS**, **Cocoa** vid utveckling av applikationer för **macOS**, etc. Dessa bibliotek är omfattande och består i sin tur av många ramverk. För att använda dessa bibliotek krävs `import`-satser, olika för varje ramverk.

Programspråket C:s standardbibliotek

Matematiska funktioner

Matematiska 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 = Double.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 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 funktioner:

- 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`. (Fr o m **Swift 3.0** kan avrundning enklare utföras med någon av instansmetoderna `rounded()` resp `round()` i flyttalstyperna.)
- 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`. (För ett bättre alternativ fr o m **Swift 4.2**, se nästa sida!)

Den matematiska konstanten π är tillgänglig i Swift via typkonstanten `Double.pi`.

Swifts standardbibliotek

Slumpmässig generering

Stöd för slumpmässig generering finns i Swifts standardbibliotek from **Swift 4.2**.

De numeriska typerna samt typen `Bool` kan generera slumpmässiga värden:

```
let randomNumber = Double.random(in: 0.0..<1.0)
                                        // Slumptal mellan 0 och 1
let diceRoll = Int.random(in: 1...6)
                                        // Slumpmässigt tärningsslag
let randomTruth = Bool.random()
                                        // Sant eller falskt
```

I Swifts datastrukturer kan ett element väljas slumpmässigt:

```
let nucleidBases = ["A", "G", "T", "C"]
if let randomBase = nucleidBases.randomElement() {
    ...                                 // Slumpmässigt valt element
}
```

Fr o m **Swift 4.2** är slumpmässig generering inbyggt i Swifts eget standardbibliotek, vilket eliminerar behovet av de något obekväma funktionerna i C:s standardbibliotek. Dessutom är slumpgeneratorn i Swifts standardbibliotek av mycket hög kvalitet.

Samtliga numeriska typer har stöd för att generera ett slumpmässigt valt värde inom ett intervall av typens värden via typmetoden `random(in:)`. T ex kan slumpmässiga flyttal i intervallet fr o m `0.0` till (men inte t o m) `1.0` genereras med

```
let randomNumber = Double.random(in: 0.0..<1.0)
```

Även slutna intervall kan användas. Ett tärningsslag (heltalsvärde i intervallet `1` t o m `6`) kan genereras med

```
let diceRoll = Int.random(in: 1...6)
```

Ett slumpmässigt värde av typen `Bool` (dvs `false` eller `true` med lika sannolikhet) kan genereras med

```
let randomTruth = Bool.random()
```

Därtill har Swifts datastrukturer förstärkts fr o m **Swift 4.2** med instansmetoden `randomElement()` som slumpmässigt väljer ett element i datastrukturen. Metodens returvärde är av en typ som kan sakna värde:

```
let nucleidBases = ["A", "G", "T", "C"]
if let randomBase = nucleidBases.randomElement() {
    print(randomBase)
}
```

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 parameter- eller returtyp för ett värde av en godtycklig typ
- Typen `AnyObject`
  - ➜ Representerar en referens till ett objekt av en godtycklig klass
  - ➜ Kompilatorn godkänner alla meddelanden som skickas via en `AnyObject`-referens
  - ➜ Motsvarar typen `id` i **Objective-C**
  - ➜ Förekommer ofta i kommunikationen med **Cocoa**- och **Cocoa Touch**-biblioteken
  - ➜ Används i biblioteken för datastrukturer av objekt utan känd typ
  - ➜ Används för att definiera gränssnitt som endast får implementeras av klasser

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 parameter- 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. En Swift-kompilator godkänner alla meddelanden som skickas till en objektreferens av typen `AnyObject` (men om mottagaren saknar en metod att exekvera, bryts exekveringen omedelbart av ett exekverings-fel). Detta är Swifts motsvarighet till typen `id` i programspråket **Objective-C**. Typen `AnyObject` förekommer ofta i deklarationer av typer för egenskaper, parametrar eller returvärden i **Cocoa**- och **Cocoa Touch**-biblioteken, i synnerhet som grundtyp för element i datastrukturer där objektens exakta klasstillhörighet inte är känd.

  Typen `AnyObject` används även för att definiera gränssnitt som endast får implementeras av klasser:

  ```
  protocol LoggingDelegate: AnyObject { … }
  ```

  Se kapitlet *Gränssnitt* för en utförligande förklaring!

Foundation-ramverket definierar därtill 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 i stil med:

```
let account1 = try! Account(no: "PS721-39", balance: 15000)
print(account1)                       // "__lldb_expr_109.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

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

    func hash(into hasher: inout Hasher) {
        hasher.combine(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 typmetod (`static func`) med namnet `==`. Metoden ska ha två parametrar av den egendefinierade typen och returtypen `Bool`:

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

Metoden 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 `==`:

```
static 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 vi automatiskt tillgång till operatorn `!=`, som jämför om två värden är olika:

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

Före **Swift 3.0** kunde en operator endast definieras som en global funktion:

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

Det är fortfarande tillåtet, men rekommendationen fr o m **Swift 3.0** är att definiera jämförelseoperatorer som typmetoder. För klasser kan de alternativt definieras som `final`-deklarerade klassmetoder:

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

En likhetsjämförelse med operatorn `==` bör alltid definieras så att den uppfyller de tre kraven för en *ekvivalensrelation* mellan värden:

- *Reflexivitet:* `a == a` ska alltid vara sant (`true`)
- *Symmetri:* om `a == b`, så ska alltid `b == a`
- *Transitivitet:* om `a == b` och `b == c`, så ska alltid `a == c`

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 metoden `hash(into:)`. Denna metod används i standardbiblioteket 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 de hashkoder som genereras för två värden av samma typ måste vara lika om operatorn `==` bedömt dem som lika. Vi behöver därför implementera metoden `hash(into:)` på ett sätt som stämmer överens med vår implementation av operatorn `==`. Lyckligtvis är detta enkelt att göra.

Metoden `hash(into:)` tar emot ett argument, en referensöverförd (dvs `inout`-deklarerad) post av posttypen `Hasher`. Denna post kapslar in en hashfunktion som genererar hashkoder av hög kvalitet, utan att vi behöver behärska detaljerna. Det enda vi behöver göra är att anropa `Hasher`-postens metod `combine(_:)` en gång för varje egenskap som hashkoden ska baseras på. Detta ska vara exakt samma egenskaper som vi byggde likhetsjämförelsen med operatorn `==` på, dvs i vårt fall endast egenskapen `accountNo`:

```
func hash(into hasher: inout Hasher) {
    hasher.combine(accountNo)
}
```

Swifts standardbibliotek

Kompilatorgenererad implementation av Hashable

Många av standardbibliotekets typer implementerar redan `Hashable`.

Fr o m **Swift 4.2** kan kompilatorn generera en `Hashable`-implementation för:

- posttyper där samtliga egenskaper implementerar `Hashable`
- uppräknade datatyper där anknutna värden implementerar `Hashable`

```
struct PostalAddress: Hashable {
    var streetAddress: String
    var postCode:      Int
    var city:          String
}
```

```
let address1 = PostalAddress(streetAddress: "Bygatan 42",
                             postCode: 98121, city: "KIRUNA")
let address2 = PostalAddress(streetAddress: "Bygatan 42",
                             postCode: 98121, city: "KIRUNA")
if address1 == address2 {
    print("Samma adress!")
}
```

De grundläggande datatyperna inklusive typen `String` samt uppräknade datatyper utan anknutna värden implementerar redan gränssnittet `Hashable`.

För alla övriga typer är det vi som väljer om de ska implementera `Hashable` eller inte, och i så fall implementera gränssnittet så som visades på föregående sidor.

Fr o m **Swift 4.2** kan emellertid kompilatorn hjälpa oss att generera en korrekt implementation i följande fall:

- för posttyper där samtliga egenskaper redan implementerar gränssnittet `Hashable`
- för uppräknade datatyper med anknutna värden där samtliga typer för de anknutna värdena implementerar gränssnittet `Hashable`

I dessa fall räcker det att vi deklarerar att vi vill att vår typ ska implementera gränssnittet:

```
struct PostalAddress: Hashable { ... }
```

så kommer kompilatorn att generera korrekt kod för både operatorn `==` och metoden `hash(into:)`.

Observera att kompilatorn aldrig kan generera implementationer av `Hashable` för klasser.

Swifts standardbibliotek

Storleksjämförelse av värden

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

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

    static 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 `ISODate`, 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 tidigare exempel. I implementationen drar vi nytta av den beräknade egenskapen `dayNumber`, som returnerar ett löpande dagnummer inom ett år:

```
static 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 typmetod med namnet <. Metoden ska ha två parametrar av den egendefinierade typen och returtypen `Bool`:

```
static 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

Iteration över värden av uppräknade datatyper

Fr o m **Swift 4.2** kan vi enkelt iterera över värden av en uppräknad datatyp:

- Den uppräknade datatypen ska implementera gränssnittet `CaseIterable`
- Om det finns en matchande typ skrivs den före gränssnittsrelationen

```
enum Weekday: String, CaseIterable {
    case monday    = "måndag"
    case tuesday   = "tisdag"
    case wednesday = "onsdag"
    case thursday  = "torsdag"
    case friday    = "fredag"
    case saturday  = "lördag"
    case sunday    = "söndag"
}
```

Egenskapen `allCases` ger oss nu en vektor av typens samtliga värden:

```
for day in Weekday.allCases {
    print(day)
}
```

Vi har tidigare sett att värden av uppräknade datatyper fungerar utmärkt med styrstrukturer som t ex `if`- och `switch`-satserna, men från början fanns det i Swift inget enkelt sätt att iterera över samtliga värden av en uppräknad datatyp.

Fr o m **Swift 4.2** kan detta lösas enkelt genom att se till att den uppräknade datatypen implementerar gränssnittet `CaseIterable`. Förbehållet är att den uppräknade datatypen inte får ha anknutna värden. Vi behöver endast deklarera att typen implementerar gränssnittet – kompilatorn genererar en korrekt implementation:

```
enum Weekday: CaseIterable { ... }
```

Om den uppräknade datatypen har matchande värden ska typen för de matchande värdena deklareras före gränssnittsrelationen:

```
enum Weekday: String, CaseIterable { ... }
```

Kompilatorn genererar implementationen av en typegenskap `allCases`, som är en vektor av samtliga värden av den uppräknade datatypen, i den ordning de deklareras i typen. Vi kan nu lätt iterera över dessa värden med en for-in-sats:

```
for day in Weekday.allCases {
    print(day)
}
```