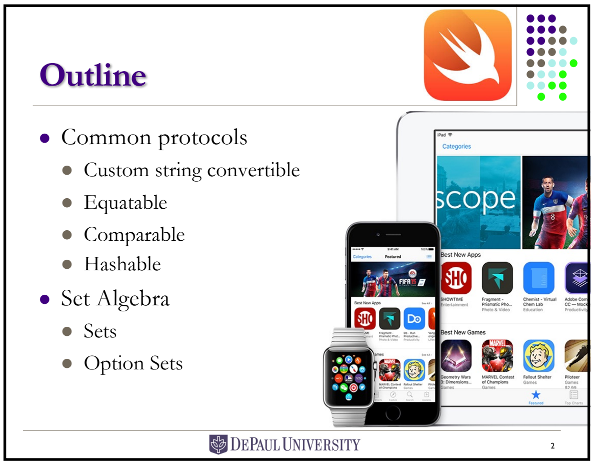
CSC 491 / 391 Mobile Application Development for iOS II

Prof. Xiaoping Jia
School of Computing, CDM
DePaul University
xjia@cdm.depaul.edu
@DePaulSWEng

1

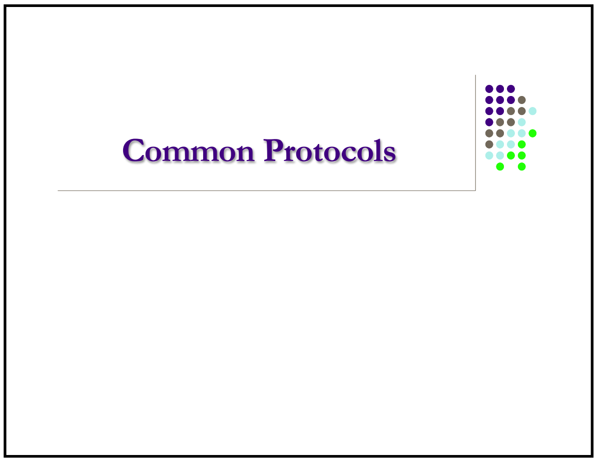


Outline

- Common protocols
  - Custom string convertible
  - Equatable
  - Comparable
  - Hashable
- Set Algebra
  - Sets
  - Option Sets

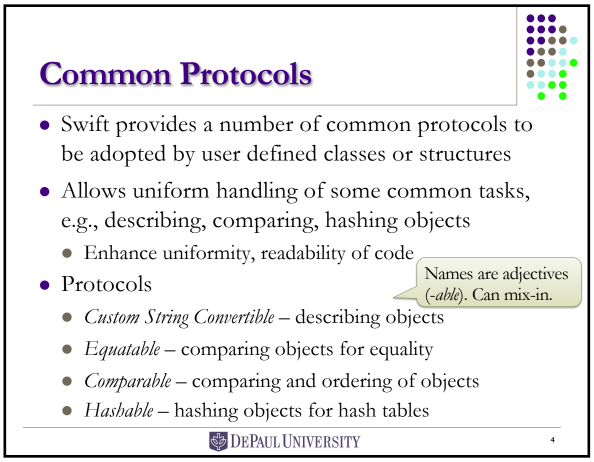
DePaul University 2

2



Common Protocols

3



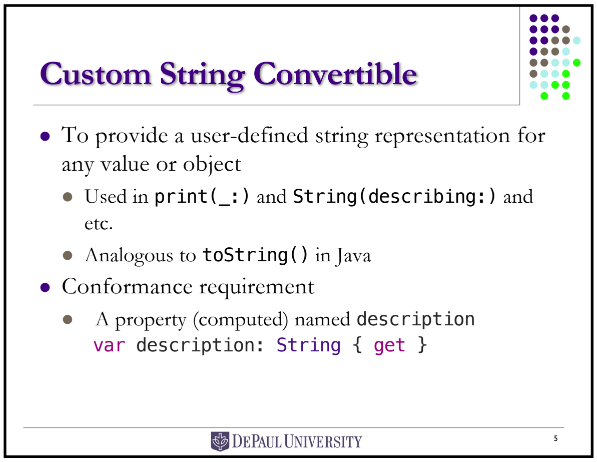
Common Protocols

- Swift provides a number of common protocols to be adopted by user defined classes or structures
- Allows uniform handling of some common tasks, e.g., describing, comparing, hashing objects
  - Enhance uniformity, readability of code
- Protocols
  - *Custom String Convertible* – describing objects
  - *Equatable* – comparing objects for equality
  - *Comparable* – comparing and ordering of objects
  - *Hashable* – hashing objects for hash tables

Names are adjectives (*-able*). Can mix-in.

DePaul University 4

4



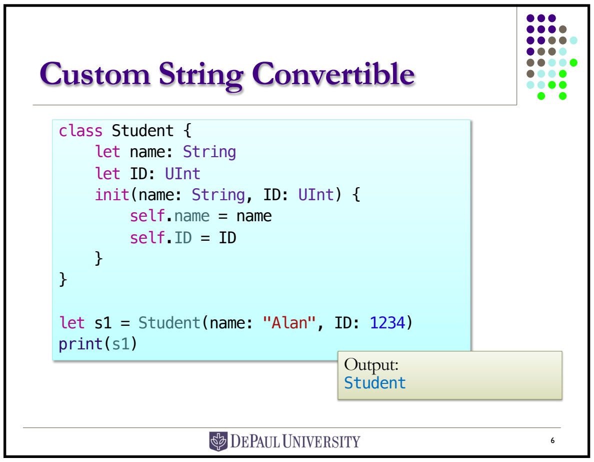
Custom String Convertible

- To provide a user-defined string representation for any value or object
  - Used in `print(_:)` and `String(describing:)` and etc.
  - Analogous to `toString()` in Java
- Conformance requirement
  - A property (computed) named `description`

```
var description: String { get }
```

DePaul University 5

5



Custom String Convertible

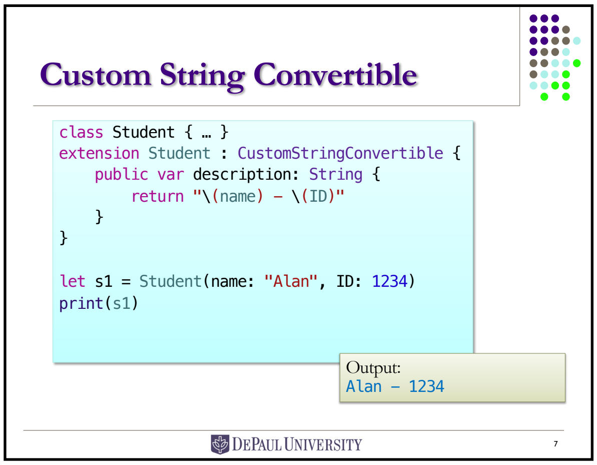
```
class Student {
    let name: String
    let ID: UInt
    init(name: String, ID: UInt) {
        self.name = name
        self.ID = ID
    }
}

let s1 = Student(name: "Alan", ID: 1234)
print(s1)
```

Output:

```
Student
```

DePaul University 6

6



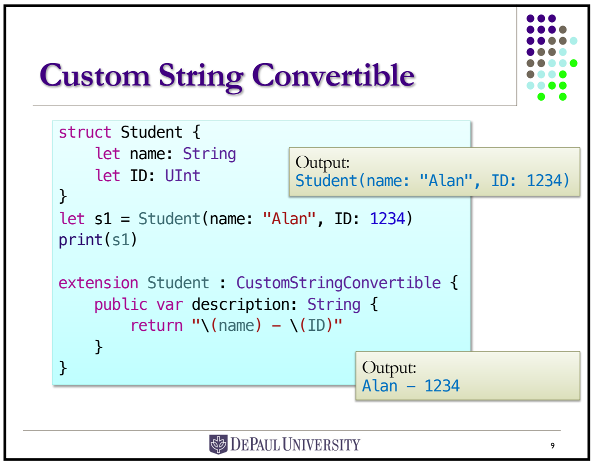
Custom String Convertible

```
class Student { … }
extension Student : CustomStringConvertible {
    public var description: String {
        return "\(name) – \(ID)"
    }
}

let s1 = Student(name: "Alan", ID: 1234)
print(s1)
```

Output:
```
Alan – 1234
```

DePaul University 7

7



Custom String Convertible

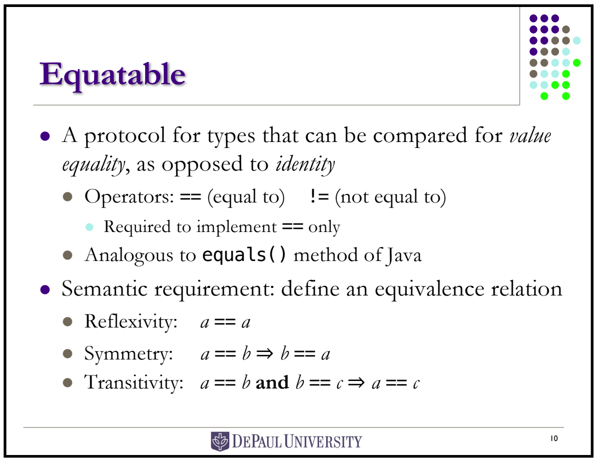
```
struct Student {
    let name: String
    let ID: UInt
}
let s1 = Student(name: "Alan", ID: 1234)
print(s1)

extension Student : CustomStringConvertible {
    public var description: String {
        return "\(name) – \(ID)"
    }
}
```

Output:
```
Student(name: "Alan", ID: 1234)
```

Output:
```
Alan – 1234
```

DePaul University 9

9

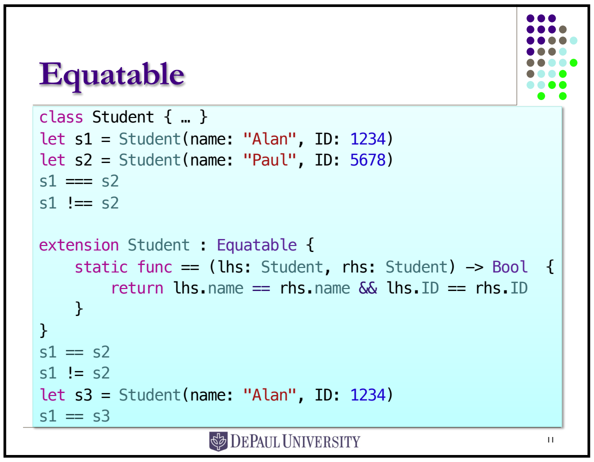


Equatable

- A protocol for types that can be compared for *value equality*, as opposed to *identity*
  - Operators: == (equal to)   != (not equal to)
    - Required to implement == only
  - Analogous to `equals()` method of Java
- Semantic requirement: define an equivalence relation
  - Reflexivity: $a == a$
  - Symmetry: $a == b \Rightarrow b == a$
  - Transitivity: $a == b$ **and** $b == c \Rightarrow a == c$

DePaul University 10

10

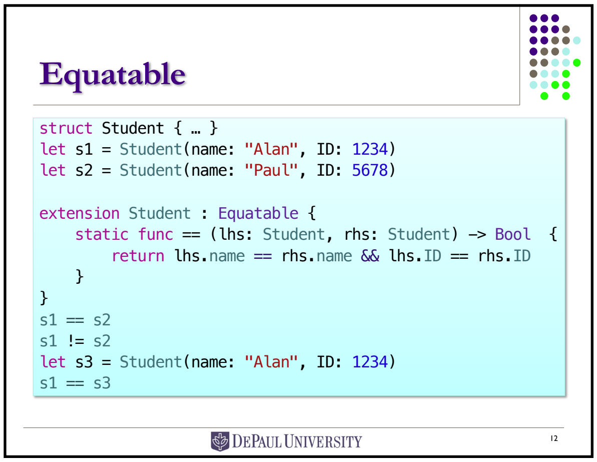


Equatable

```
class Student { … }
let s1 = Student(name: "Alan", ID: 1234)
let s2 = Student(name: "Paul", ID: 5678)
s1 === s2
s1 !== s2

extension Student : Equatable {
    static func == (lhs: Student, rhs: Student) -> Bool  {
        return lhs.name == rhs.name && lhs.ID == rhs.ID
    }
}
s1 == s2
s1 != s2
let s3 = Student(name: "Alan", ID: 1234)
s1 == s3
```

DePaul University 11

11

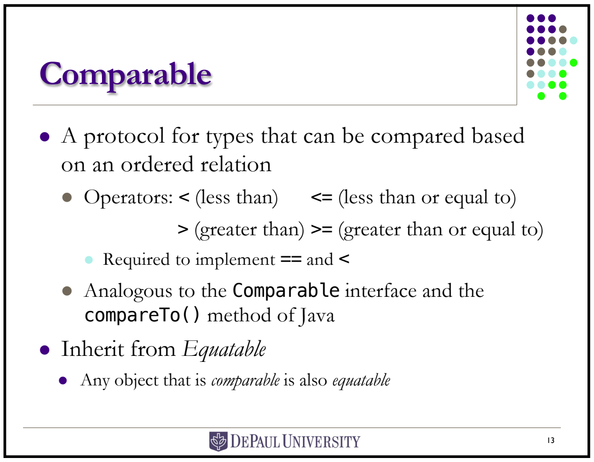


Equatable

```
struct Student { … }
let s1 = Student(name: "Alan", ID: 1234)
let s2 = Student(name: "Paul", ID: 5678)

extension Student : Equatable {
    static func == (lhs: Student, rhs: Student) -> Bool  {
        return lhs.name == rhs.name && lhs.ID == rhs.ID
    }
}
s1 == s2
s1 != s2
let s3 = Student(name: "Alan", ID: 1234)
s1 == s3
```

DePaul University 12

12

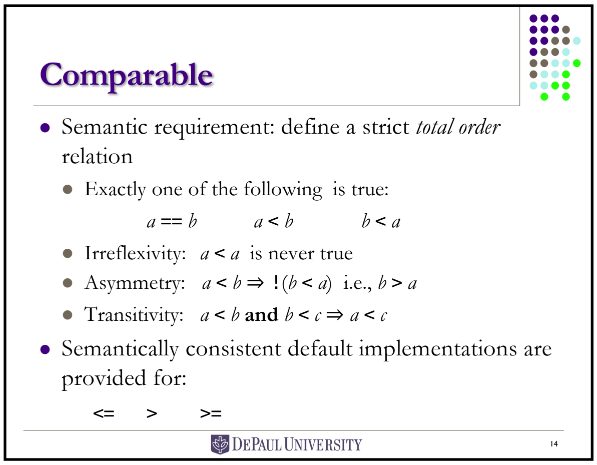


Comparable

- A protocol for types that can be compared based on an ordered relation
  - Operators: < (less than)   <= (less than or equal to)
    > (greater than) >= (greater than or equal to)
    - Required to implement == and <
  - Analogous to the `Comparable` interface and the `compareTo()` method of Java
- Inherit from *Equatable*
  - Any object that is *comparable* is also *equatable*

DePaul University 13

13



Comparable

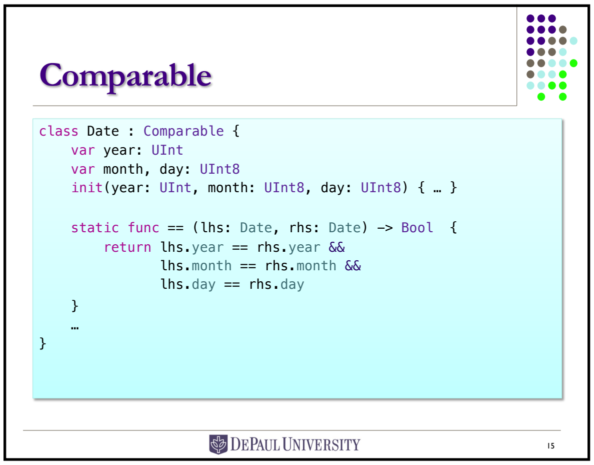
- Semantic requirement: define a strict *total order* relation
  - Exactly one of the following is true:

    $a == b$ $\quad a < b \quad$ $b < a$
  - Irreflexivity: $a < a$ is never true
  - Asymmetry: $a < b \Rightarrow !(b < a)$ i.e., $b > a$
  - Transitivity: $a < b$ **and** $b < c \Rightarrow a < c$
- Semantically consistent default implementations are provided for:

  <= > >=

DePaul University 14

14

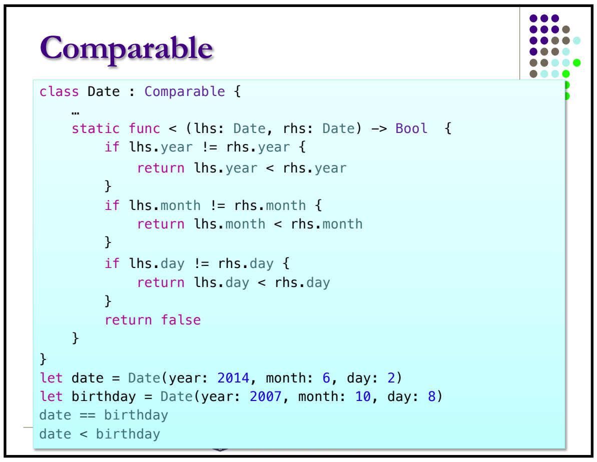


Comparable

```
class Date : Comparable {
    var year: UInt
    var month, day: UInt8
    init(year: UInt, month: UInt8, day: UInt8) { … }

    static func == (lhs: Date, rhs: Date) -> Bool  {
        return lhs.year == rhs.year &&
               lhs.month == rhs.month &&
               lhs.day == rhs.day
    }
    …
}
```

DePaul University 15

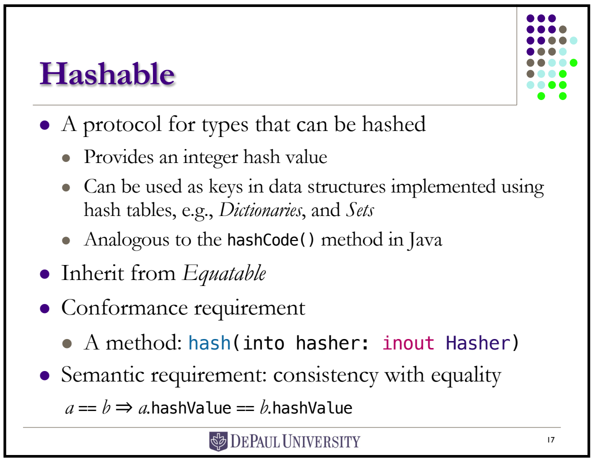
15



Comparable

```
class Date : Comparable {
    …
    static func < (lhs: Date, rhs: Date) -> Bool  {
        if lhs.year != rhs.year {
            return lhs.year < rhs.year
        }
        if lhs.month != rhs.month {
            return lhs.month < rhs.month
        }
        if lhs.day != rhs.day {
            return lhs.day < rhs.day
        }
        return false
    }
}
let date = Date(year: 2014, month: 6, day: 2)
let birthday = Date(year: 2007, month: 10, day: 8)
date == birthday
date < birthday
```

16



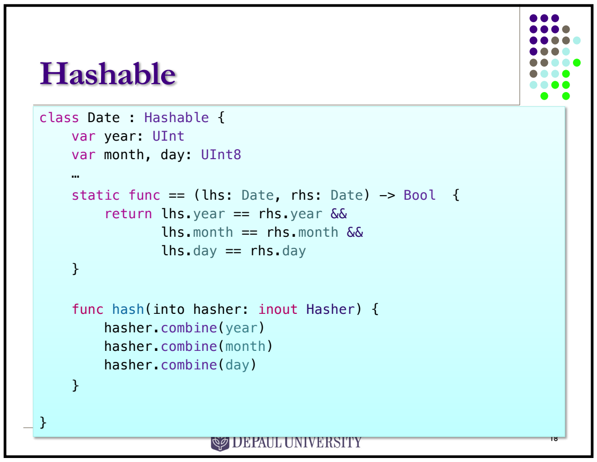
Hashable

- A protocol for types that can be hashed
  - Provides an integer hash value
  - Can be used as keys in data structures implemented using hash tables, e.g., *Dictionaries*, and *Sets*
  - Analogous to the `hashCode()` method in Java
- Inherit from *Equatable*
- Conformance requirement
  - A method: `hash(into hasher: inout Hasher)`
- Semantic requirement: consistency with equality

  $a == b \Rightarrow a.\mathtt{hashValue} == b.\mathtt{hashValue}$

DePaul University 17

17

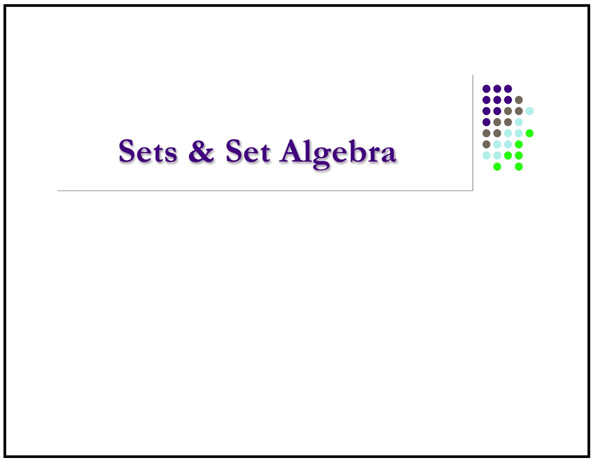


Hashable

```
class Date : Hashable {
    var year: UInt
    var month, day: UInt8
    …
    static func == (lhs: Date, rhs: Date) -> Bool  {
        return lhs.year == rhs.year &&
               lhs.month == rhs.month &&
               lhs.day == rhs.day
    }

    func hash(into hasher: inout Hasher) {
        hasher.combine(year)
        hasher.combine(month)
        hasher.combine(day)
    }

}
```

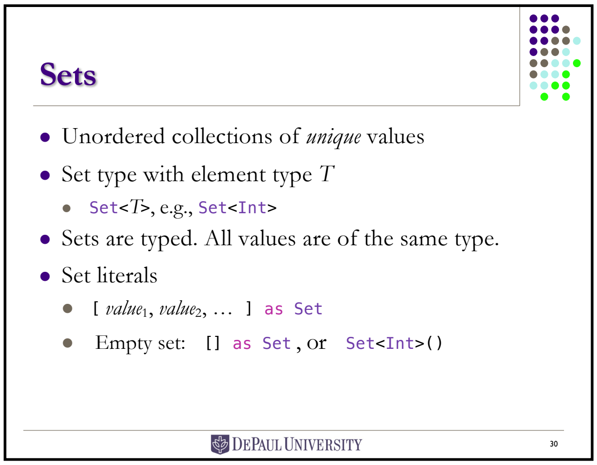
DePaul University 18

18



Sets & Set Algebra

19

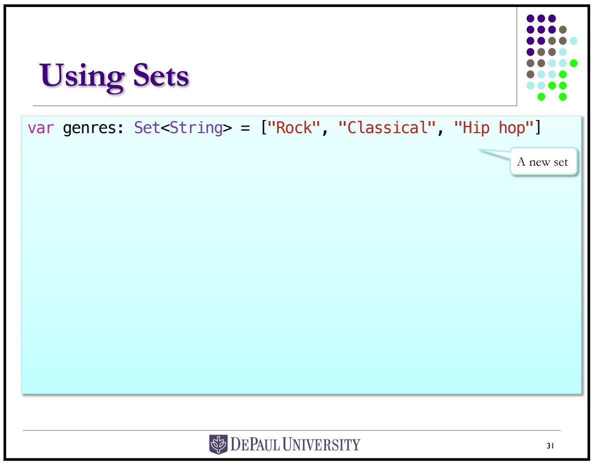


Sets

- Unordered collections of *unique* values
- Set type with element type *T*
  - `Set<`*T*`>`, e.g., `Set<Int>`
- Sets are typed. All values are of the same type.
- Set literals
  - `[` *value*$_1$, *value*$_2$, … `] as Set`
  - Empty set: `[] as Set` , or `Set<Int>()`

DePaul University 30

20



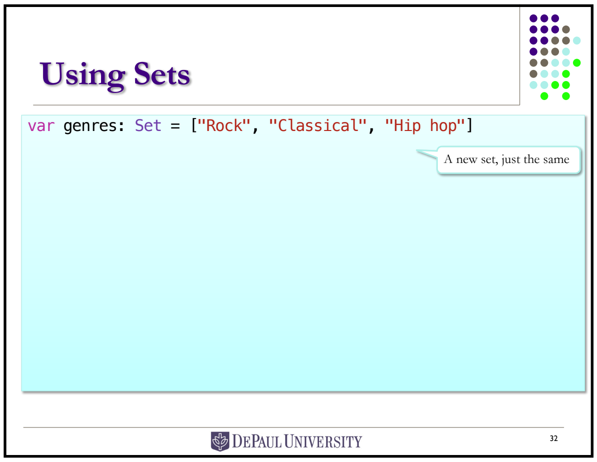
Using Sets

```
var genres: Set<String> = ["Rock", "Classical", "Hip hop"]
```

A new set

DePaul University 31

21



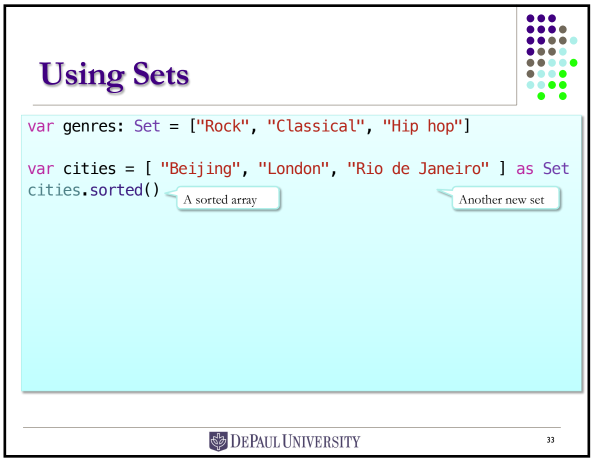
Using Sets

```
var genres: Set = ["Rock", "Classical", "Hip hop"]
```

A new set, just the same

DePaul University 32

22



Using Sets

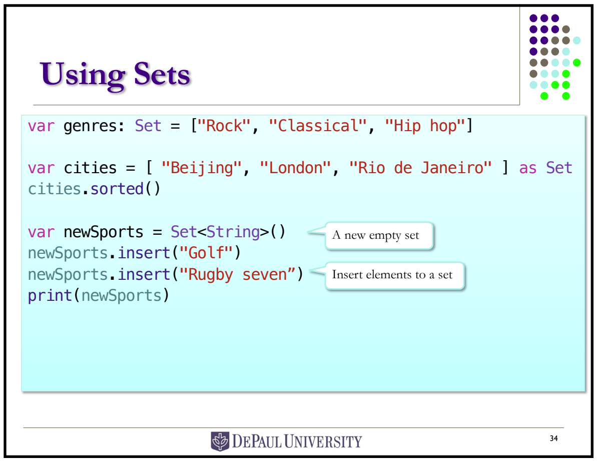
```
var genres: Set = ["Rock", "Classical", "Hip hop"]

var cities = [ "Beijing", "London", "Rio de Janeiro" ] as Set
cities.sorted()
```

A sorted array

Another new set

DePaul University 33

23



Using Sets

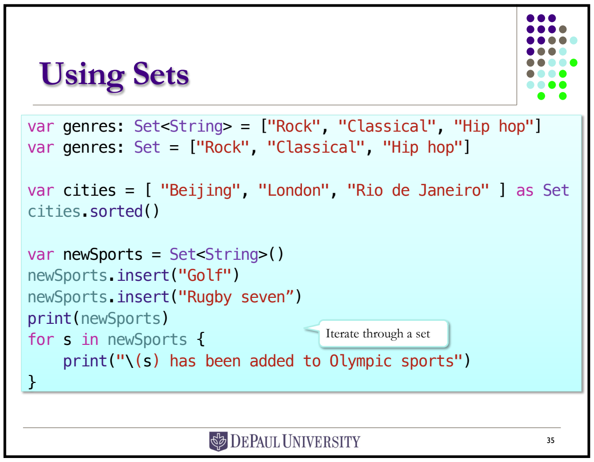
```
var genres: Set = ["Rock", "Classical", "Hip hop"]

var cities = [ "Beijing", "London", "Rio de Janeiro" ] as Set
cities.sorted()

var newSports = Set<String>()
newSports.insert("Golf")
newSports.insert("Rugby seven”)
print(newSports)
```

A new empty set

Insert elements to a set

DePaul University 34

24



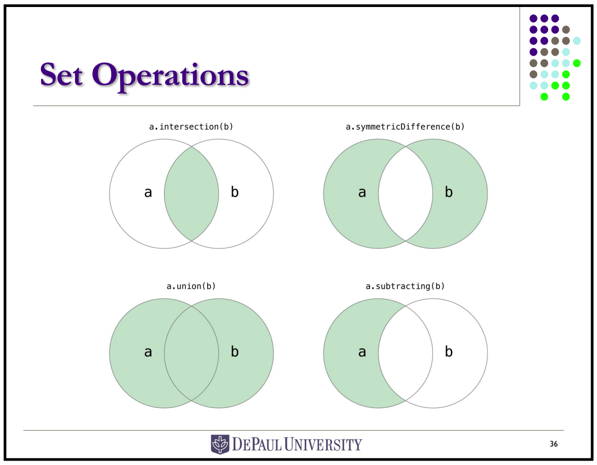
Using Sets

```
var genres: Set<String> = ["Rock", "Classical", "Hip hop"]
var genres: Set = ["Rock", "Classical", "Hip hop"]

var cities = [ "Beijing", "London", "Rio de Janeiro" ] as Set
cities.sorted()

var newSports = Set<String>()
newSports.insert("Golf")
newSports.insert("Rugby seven”)
print(newSports)
for s in newSports {
    print("\(s) has been added to Olympic sports")
}
```

Iterate through a set

DePaul University 35

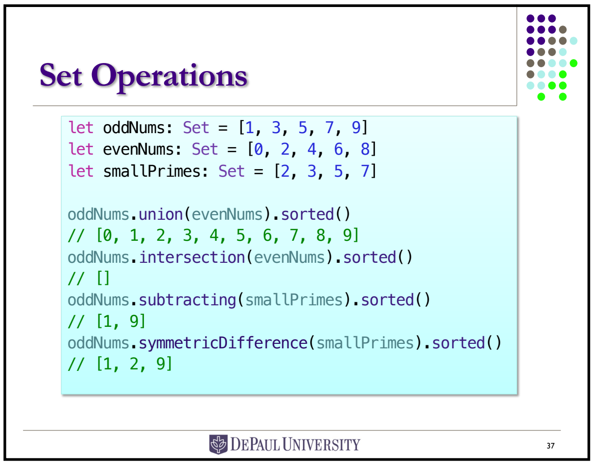
25



Set Operations

DePaul University 36

26

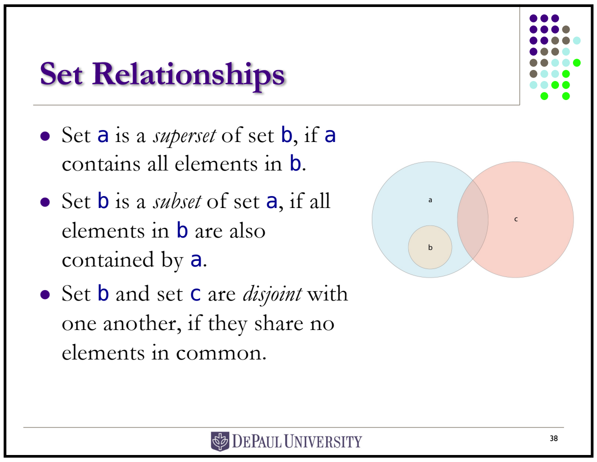


Set Operations

```
let oddNums: Set = [1, 3, 5, 7, 9]
let evenNums: Set = [0, 2, 4, 6, 8]
let smallPrimes: Set = [2, 3, 5, 7]

oddNums.union(evenNums).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddNums.intersection(evenNums).sorted()
// []
oddNums.subtracting(smallPrimes).sorted()
// [1, 9]
oddNums.symmetricDifference(smallPrimes).sorted()
// [1, 2, 9]
```

DePaul University 37

27

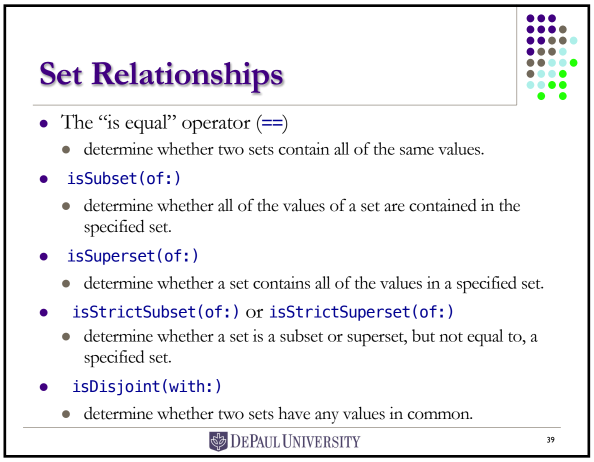


Set Relationships

- Set `a` is a *superset* of set `b`, if `a` contains all elements in `b`.
- Set `b` is a *subset* of set `a`, if all elements in `b` are also contained by `a`.
- Set `b` and set `c` are *disjoint* with one another, if they share no elements in common.

DePaul University 38

28

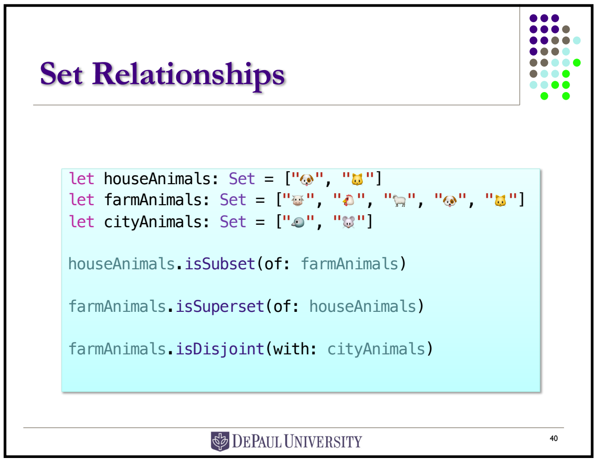


Set Relationships

- The "is equal" operator (`==`)
  - determine whether two sets contain all of the same values.
- `isSubset(of:)`
  - determine whether all of the values of a set are contained in the specified set.
- `isSuperset(of:)`
  - determine whether a set contains all of the values in a specified set.
- `isStrictSubset(of:)` or `isStrictSuperset(of:)`
  - determine whether a set is a subset or superset, but not equal to, a specified set.
- `isDisjoint(with:)`
  - determine whether two sets have any values in common.

DePaul University 39

29

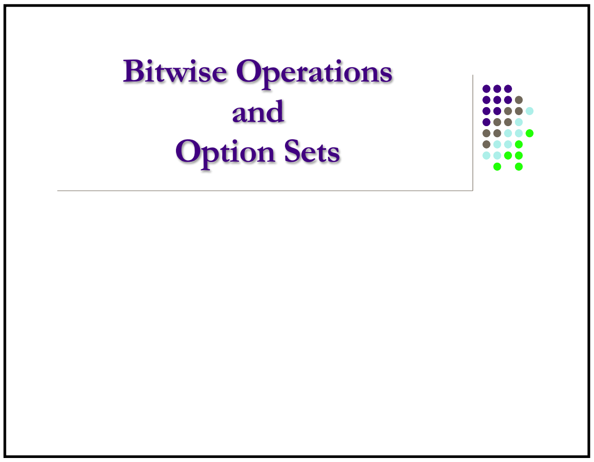


Set Relationships

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐦", "🐭"]

houseAnimals.isSubset(of: farmAnimals)

farmAnimals.isSuperset(of: houseAnimals)

farmAnimals.isDisjoint(with: cityAnimals)
```

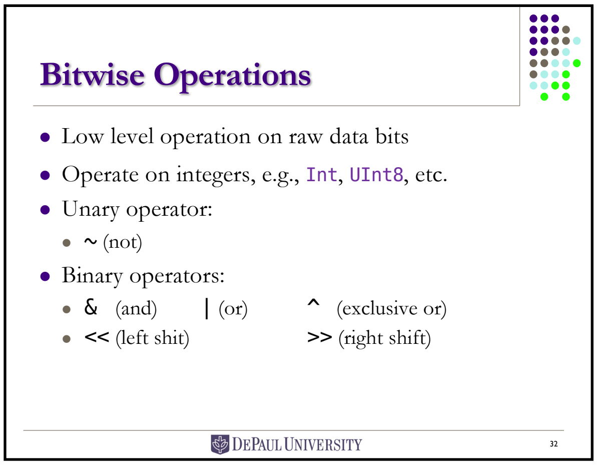
DePaul University 40

30



Bitwise Operations and Option Sets

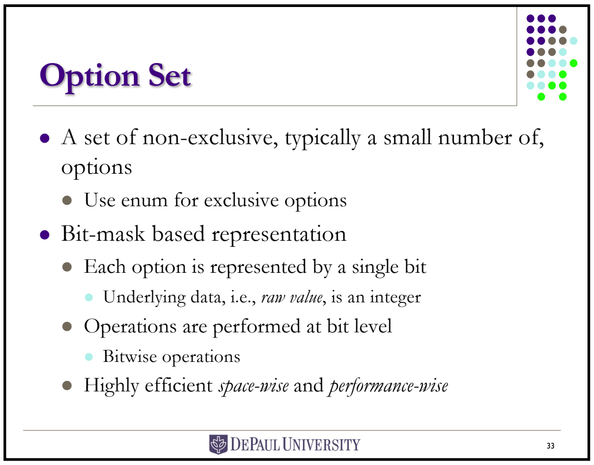
31



Bitwise Operations

- Low level operation on raw data bits
- Operate on integers, e.g., `Int`, `UInt8`, etc.
- Unary operator:
  - ~ (not)
- Binary operators:
  - & (and) | (or) ^ (exclusive or)
  - << (left shit) >> (right shift)

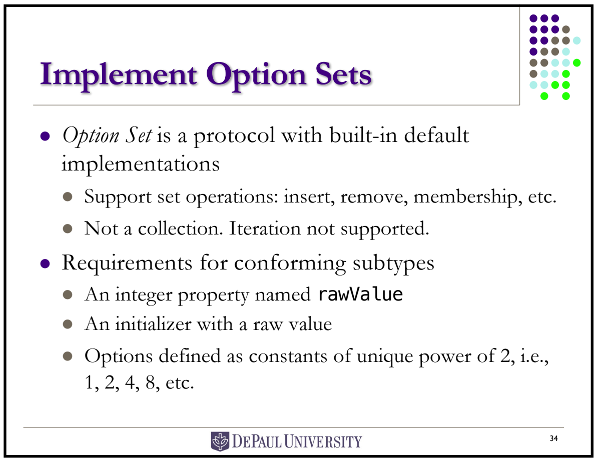
32



Option Set

- A set of non-exclusive, typically a small number of, options
  - Use enum for exclusive options
- Bit-mask based representation
  - Each option is represented by a single bit
    - Underlying data, i.e., *raw value*, is an integer
  - Operations are performed at bit level
    - Bitwise operations
  - Highly efficient *space-wise* and *performance-wise*

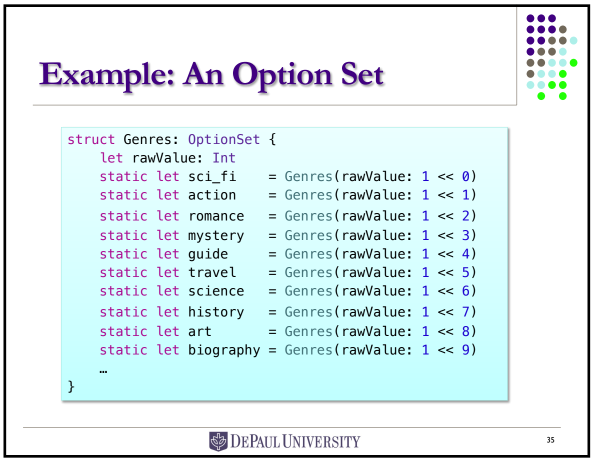
33



Implement Option Sets

- *Option Set* is a protocol with built-in default implementations
  - Support set operations: insert, remove, membership, etc.
  - Not a collection. Iteration not supported.
- Requirements for conforming subtypes
  - An integer property named `rawValue`
  - An initializer with a raw value
  - Options defined as constants of unique power of 2, i.e., 1, 2, 4, 8, etc.

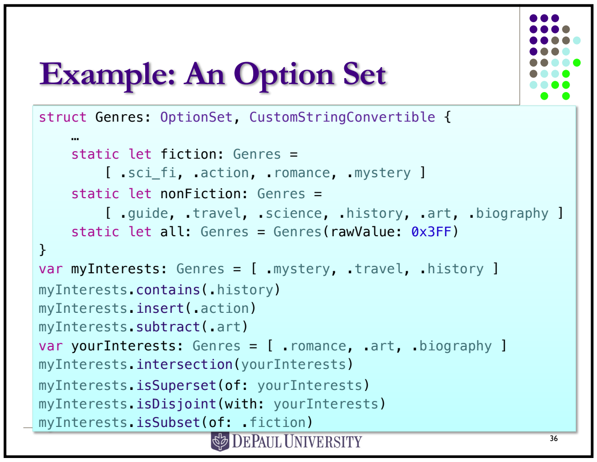
34



Example: An Option Set

```
struct Genres: OptionSet {
    let rawValue: Int
    static let sci_fi    = Genres(rawValue: 1 << 0)
    static let action    = Genres(rawValue: 1 << 1)
    static let romance   = Genres(rawValue: 1 << 2)
    static let mystery   = Genres(rawValue: 1 << 3)
    static let guide     = Genres(rawValue: 1 << 4)
    static let travel    = Genres(rawValue: 1 << 5)
    static let science   = Genres(rawValue: 1 << 6)
    static let history   = Genres(rawValue: 1 << 7)
    static let art       = Genres(rawValue: 1 << 8)
    static let biography = Genres(rawValue: 1 << 9)
    …
}
```

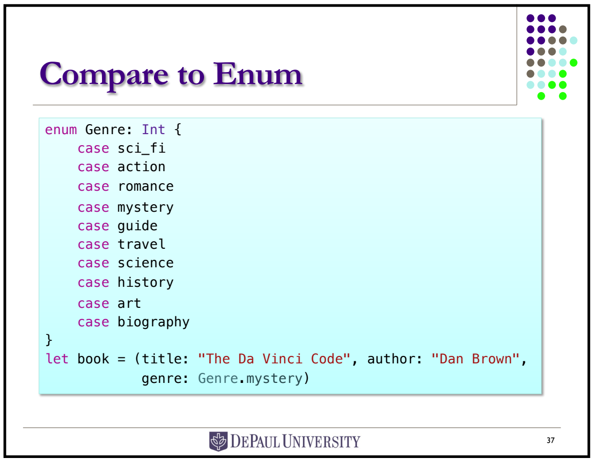
35



Example: An Option Set

```
struct Genres: OptionSet, CustomStringConvertible {
    …
    static let fiction: Genres =
        [ .sci_fi, .action, .romance, .mystery ]
    static let nonFiction: Genres =
        [ .guide, .travel, .science, .history, .art, .biography ]
    static let all: Genres = Genres(rawValue: 0x3FF)
}
var myInterests: Genres = [ .mystery, .travel, .history ]
myInterests.contains(.history)
myInterests.insert(.action)
myInterests.subtract(.art)
var yourInterests: Genres = [ .romance, .art, .biography ]
myInterests.intersection(yourInterests)
myInterests.isSuperset(of: yourInterests)
myInterests.isDisjoint(with: yourInterests)
myInterests.isSubset(of: .fiction)
```

36



Compare to Enum

```
enum Genre: Int {
    case sci_fi
    case action
    case romance
    case mystery
    case guide
    case travel
    case science
    case history
    case art
    case biography
}
let book = (title: "The Da Vinci Code", author: "Dan Brown",
            genre: Genre.mystery)
```

37