Chapter 21: Type Casting

Section 21.1: Downcasting

A variable can be downcasted to a subtype using the *type cast operators* `as?`, and `as!`.

The `as?` operator *attempts* to cast to a subtype.
It can fail, therefore it returns an optional.

```
let value: Any = "John"

let name = value as? String
print(name) // prints Optional("John")

let age = value as? Double
print(age) // prints nil
```

The `as!` operator *forces* a cast.
It does not return an optional, but crashes if the cast fails.

```
let value: Any = "Paul"

let name = value as! String
print(name) // prints "Paul"

let age = value as! Double // crash: "Could not cast value..."
```

It is common to use type cast operators with conditional unwrapping:

```
let value: Any = "George"

if let name = value as? String {
    print(name) // prints "George"
}

if let age = value as? Double {
    print(age) // Not executed
}
```

Section 21.2: Type casting in Swift Language

Type Casting

Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

Type casting in Swift is implemented with the is and as operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.

Downcasting

A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to downcast to the subclass type with a type cast operator (as? or

as!).

Because downcasting can fail, the type cast operator comes in two different forms. The conditional form, as?, returns an optional value of the type you are trying to downcast to. The forced form, as!, attempts the downcast and force-unwraps the result as a single compound action.

Use the conditional form of the type cast operator (as?) when you are not sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be nil if the downcast was not possible. This enables you to check for a successful downcast.

Use the forced form of the type cast operator (as!) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type. Know more.

String to Int & Float conversion : -

```
    let numbers = "888.00"
    let intValue = NSString(string: numbers).integerValue
    print(intValue) // Output - 888


    let numbers = "888.00"
    let floatValue = NSString(string: numbers).floatValue
    print(floatValue) // Output : 888.0
```

Float to String Conversion

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue) // Output : 888.0

    // Get Float value at particular decimal point
    let numbers = 888.00
    let floatValue = String(format: "%.2f", numbers) // Here %.2f will give 2 numbers after decimal
points we can use as per our need
    print(floatValue) // Output : "888.00"
```

Integer to String value

```
    let numbers = 888
    let intValue = String(numbers)
    print(intValue) // Output : "888"
```

Float to String value

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue)
```

Optional Float value to String

```
    let numbers: Any = 888.00
    let floatValue = String(describing: numbers)
    print(floatValue) // Output : 888.0
```

Optional String to Int value

```
    let hitCount = "100"
    let data :AnyObject = hitCount
    let score = Int(data as? String ?? "") ?? 0
    print(score)
```

Downcasting values from JSON

```
    let json = ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]] as
[String : Any]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Downcasting values from Optional JSON

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]]
    let json = response as? [String: Any] ?? [:]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Manage JSON Response with conditions

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]] //Optional Response

    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
    }
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Manage Nil Response with condition

```
    let response: Any? = nil
    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
    }
    let name = json["name"] as? String ?? ""
    print(name)
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects)
```

Output : `Empty Dictionary`

Section 21.3: Upcasting

The `as` operator will cast to a supertype. As it cannot fail, it does not return an optional.

```
let name = "Ringo"
let value = string as Any  // `value` is of type `Any` now
```

Section 21.4: Example of using a downcast on a function

parameter involving subclassing

A downcast can be used to make use of a subclass's code and data inside of a function taking a parameter of its superclass.

```
class Rat {
    var color = "white"
}

class PetRat: Rat {
    var name = "Spot"
}

func nameOfRat(☐: Rat) -> String {
    guard let petRat = (☐ as? PetRat) else {
        return "No name"
    }

    return petRat.name
}

let noName = Rat()
let spot = PetRat()

print(nameOfRat(noName))
print(nameOfRat(spot))
```

Section 21.5: Casting with switch

The `switch` statement can also be used to attempt casting into different types:

```
func checkType(_ value: Any) -> String {
    switch value {

    // The `is` operator can be used to check a type
    case is Double:
        return "value is a Double"

    // The `as` operator will cast. You do not need to use `as?` in a `switch`.
    case let string as String:
        return "value is the string: \(string)"

    default:
        return "value is something else"
    }

}

checkType("Cadena")  // "value is the string: Cadena"
checkType(6.28)      // "value is a Double"
checkType(UILabel()) // "value is something else"
```