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.

You can also use type casting to check whether a type conforms to a protocol, as described in Checking for Protocol Conformance.

Defining a Class Hierarchy for Type Casting

You can use type casting with a hierarchy of classes and subclasses to check the type of a particular class instance and to cast that instance to another class within the same hierarchy. The three code snippets below define a hierarchy of classes and an array containing instances of those classes, for use in an example of type casting.

The first snippet defines a new base class called `MediaItem`. This class provides basic functionality for any kind of item that appears in a digital media library. Specifically, it declares a `name` property of type `String`, and an `init name` initializer. (It is assumed that all media items, including all movies and songs, will have a name.)

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

The next snippet defines two subclasses of `MediaItem`. The first subclass, `Movie`, encapsulates additional information about a movie or film. It adds a `director` property on top of the base `MediaItem` class, with a corresponding initializer. The second subclass, `Song`, adds an `artist` property and initializer on top of the base class:

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

The final snippet creates a constant array called `library`, which contains two `Movie` instances and three `Song` instances. The type of the `library` array is inferred by initializing it with the contents of an array literal. Swift's type checker is able to deduce that `Movie` and `Song` have a common superclass of `MediaItem`, and so it infers a type of `[MediaItem]` for the `library` array:

```
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
```

```
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
// the type of "library" is inferred to be [MediaItem]
```

The items stored in `library` are still `Movie` and `Song` instances behind the scenes. However, if you iterate over the contents of this array, the items you receive back are typed as `MediaItem`, and not as `Movie` or `Song`. In order to work with them as their native type, you need to *check* their type, or *downcast* them to a different type, as described below.

Checking Type

Use the *type check operator* (`is`) to check whether an instance is of a certain subclass type. The type check operator returns `true` if the instance is of that subclass type and `false` if it is not.

The example below defines two variables, `movieCount` and `songCount`, which count the number of `Movie` and `Song` instances in the `library` array:

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

print("Media library contains \(movieCount) movies and \(songCount) songs")
// Prints "Media library contains 2 movies and 3 songs"
```

This example iterates through all items in the `library` array. On each pass, the `for`-`in` loop sets the `item` constant to the next `MediaItem` in the array.

`item is Movie` returns `true` if the current `MediaItem` is a `Movie` instance and `false` if it is not. Similarly, `item is Song` checks whether the item is a `Song` instance. At the end of the `for`-`in` loop, the values of `movieCount` and `songCount` contain a count of how many `MediaItem` instances were found of each 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.

The example below iterates over each `MediaItem` in `library`, and prints an appropriate description for each item. To do this, it needs to access each item as a true `Movie` or `Song`, and not just as a `MediaItem`. This is necessary in order for it to be able to access the `director` or `artist` property of a `Movie` or `Song` for use in the description.

In this example, each item in the array might be a `Movie`, or it might be a `Song`. You don’t know in advance

which actual class to use for each item, and so it is appropriate to use the conditional form of the type cast operator (`as?`) to check the downcast each time through the loop:

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}

// Movie: Casablanca, dir. Michael Curtiz
// Song: Blue Suede Shoes, by Elvis Presley
// Movie: Citizen Kane, dir. Orson Welles
// Song: The One And Only, by Chesney Hawkes
// Song: Never Gonna Give You Up, by Rick Astley
```

The example starts by trying to downcast the current `item` as a `Movie`. Because `item` is a `MediaItem` instance, it's possible that it *might* be a `Movie`; equally, it's also possible that it might be a `Song`, or even just a base `MediaItem`. Because of this uncertainty, the `as?` form of the type cast operator returns an *optional* value when attempting to downcast to a subclass type. The result of `item as? Movie` is of type `Movie?`, or "optional `Movie`".

Downcasting to `Movie` fails when applied to the `Song` instances in the library array. To cope with this, the example above uses optional binding to check whether the optional `Movie` actually contains a value (that is, to find out whether the downcast succeeded.) This optional binding is written "`if let movie = item as? Movie`", which can be read as:

"Try to access `item` as a `Movie`. If this is successful, set a new temporary constant called `movie` to the value stored in the returned optional `Movie`."

If the downcasting succeeds, the properties of `movie` are then used to print a description for that `Movie` instance, including the name of its `director`. A similar principle is used to check for `Song` instances, and to print an appropriate description (including `artist` name) whenever a `Song` is found in the library.

> NOTE
>
> Casting does not actually modify the instance or change its values. The underlying instance remains the same; it is simply treated and accessed as an instance of the type to which it has been cast.

Type Casting for Any and AnyObject

Swift provides two special types for working with nonspecific types:

- `Any` can represent an instance of any type at all, including function types.
- `AnyObject` can represent an instance of any class type.

Use `Any` and `AnyObject` only when you explicitly need the behavior and capabilities they provide. It is always better to be specific about the types you expect to work with in your code.

Here's an example of using `Any` to work with a mix of different types, including function types and non-class types. The example creates an array called `things`, which can store values of type `Any`:

```
var things = [Any]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
```

```swift
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
things.append({ (name: String) -> String in "Hello, \(name)" })
```

The `things` array contains two `Int` values, two `Double` values, a `String` value, a tuple of type `(Double, Double)`, the movie “Ghostbusters”, and a closure expression that takes a `String` value and returns another `String` value.

To discover the specific type of a constant or variable that is known only to be of type `Any` or `AnyObject`, you can use an `is` or `as` pattern in a `switch` statement’s cases. The example below iterates over the items in the `things` array and queries the type of each item with a `switch` statement. Several of the `switch` statement’s cases bind their matched value to a constant of the specified type to enable its value to be printed:

```swift
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Michael"))
    default:
        print("something else")
    }
}

// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called Ghostbusters, dir. Ivan Reitman
// Hello, Michael
```

NOTE

The `Any` type represents values of any type, including optional types. Swift gives you a warning if you use an optional value where a value of type `Any` is expected. If you really do need to use an optional value as an `Any` value, you can use the `as` operator to explicitly cast the optional to `Any`, as shown below.

```swift
let optionalNumber: Int? = 3
things.append(optionalNumber)        // Warning
things.append(optionalNumber as Any) // No warning
```