Enumerations

An *enumeration* defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

If you are familiar with C, you will know that C enumerations assign related names to a set of integer values. Enumerations in Swift are much more flexible, and do not have to provide a value for each case of the enumeration. If a value (known as a “raw” value) *is* provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type.

Alternatively, enumeration cases can specify associated values of *any* type to be stored along with each different case value, much as unions or variants do in other languages. You can define a common set of related cases as part of one enumeration, each of which has a different set of values of appropriate types associated with it.

Enumerations in Swift are first-class types in their own right. They adopt many features traditionally supported only by classes, such as computed properties to provide additional information about the enumeration’s current value, and instance methods to provide functionality related to the values the enumeration represents. Enumerations can also define initializers to provide an initial case value; can be extended to expand their functionality beyond their original implementation; and can conform to protocols to provide standard functionality.

For more on these capabilities, see Properties, Methods, Initialization, Extensions, and Protocols.

Enumeration Syntax

You introduce enumerations with the `enum` keyword and place their entire definition within a pair of braces:

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

Here’s an example for the four main points of a compass:

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

The values defined in an enumeration (such as `north`, `south`, `east`, and `west`) are its *enumeration cases*. You use the `case` keyword to introduce new enumeration cases.

NOTE

Unlike C and Objective-C, Swift enumeration cases are not assigned a default integer value when they are created. In the `CompassPoint` example above, `north`, `south`, `east` and `west` do not implicitly equal `0`, `1`, `2` and `3`. Instead, the different enumeration cases are fully-fledged values in their own right, with an explicitly-defined type of `CompassPoint`.

Multiple cases can appear on a single line, separated by commas:

```
enum Planet {
    case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune
}
```

Each enumeration definition defines a brand new type. Like other types in Swift, their names (such as `CompassPoint` and `Planet`) should start with a capital letter. Give enumeration types singular rather than plural names, so that they read as self-evident:

```
var directionToHead = CompassPoint.west
```

The type of `directionToHead` is inferred when it is initialized with one of the possible values of `CompassPoint`. Once `directionToHead` is declared as a `CompassPoint`, you can set it to a different `CompassPoint` value using a shorter dot syntax:

```
directionToHead = .east
```

The type of `directionToHead` is already known, and so you can drop the type when setting its value. This makes for highly readable code when working with explicitly-typed enumeration values.

Matching Enumeration Values with a Switch Statement

You can match individual enumeration values with a `switch` statement:

```
directionToHead = .south
switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}
// Prints "Watch out for penguins"
```

You can read this code as:

“Consider the value of `directionToHead`. In the case where it equals `.north`, print `"Lots of planets have a north"`. In the case where it equals `.south`, print `"Watch out for penguins"`.”

…and so on.

As described in Control Flow, a `switch` statement must be exhaustive when considering an enumeration’s cases. If the `case` for `.west` is omitted, this code does not compile, because it does not consider the complete list of `CompassPoint` cases. Requiring exhaustiveness ensures that enumeration cases are not accidentally omitted.

When it is not appropriate to provide a `case` for every enumeration case, you can provide a `default` case to cover any cases that are not addressed explicitly:

```
let somePlanet = Planet.earth
switch somePlanet {
case .earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// Prints "Mostly harmless"
```

Associated Values

The examples in the previous section show how the cases of an enumeration are a defined (and typed) value in their own right. You can set a constant or variable to `Planet.earth`, and check for this value later. However, it is sometimes useful to be able to store *associated values* of other types alongside these case values. This enables you to store additional custom information along with the case value, and permits this information to vary each time you use that case in your code.

You can define Swift enumerations to store associated values of any given type, and the value types can be different for each case of the enumeration if needed. Enumerations similar to these are known as *discriminated unions*, *tagged unions*, or *variants* in other programming languages.

For example, suppose an inventory tracking system needs to track products by two different types of barcode. Some products are labeled with 1D barcodes in UPC format, which uses the numbers `0` to `9`. Each barcode has a "number system" digit, followed by five "manufacturer code" digits and five "product code" digits. These are followed by a "check" digit to verify that the code has been scanned correctly:

Other products are labeled with 2D barcodes in QR code format, which can use any ISO 8859-1 character and can encode a string up to 2,953 characters long:

It would be convenient for an inventory tracking system to be able to store UPC barcodes as a tuple of four integers, and QR code barcodes as a string of any length.

In Swift, an enumeration to define product barcodes of either type might look like this:

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

This can be read as:

"Define an enumeration type called `Barcode`, which can take either a value of `upc` with an associated value of type (`Int`, `Int`, `Int`, `Int`), or a value of `qrCode` with an associated value of type `String`."

This definition does not provide any actual `Int` or `String` values—it just defines the *type* of associated values that `Barcode` constants and variables can store when they are equal to `Barcode.upc` or `Barcode.qrCode`.

New barcodes can then be created using either type:

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

This example creates a new variable called `productBarcode` and assigns it a value of `Barcode.upc` with an associated tuple value of `(8, 85909, 51226, 3)`.

The same product can be assigned a different type of barcode:

```
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

At this point, the original `Barcode.upc` and its integer values are replaced by the new `Barcode.qrCode` and its string value. Constants and variables of type `Barcode` can store either a `.upc` or a `.qrCode` (together with their associated values), but they can only store one of them at any given time.

The different barcode types can be checked using a switch statement, as before. This time, however, the associated values can be extracted as part of the switch statement. You extract each associated value as a

constant (with the `let` prefix) or a variable (with the `var` prefix) for use within the `switch` case's body:

```
switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product, let check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

If all of the associated values for an enumeration case are extracted as constants, or if all are extracted as variables, you can place a single `var` or `let` annotation before the case name, for brevity:

```
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

Raw Values

The barcode example in Associated Values shows how cases of an enumeration can declare that they store associated values of different types. As an alternative to associated values, enumeration cases can come prepopulated with default values (called *raw values*), which are all of the same type.

Here's an example that stores raw ASCII values alongside named enumeration cases:

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

Here, the raw values for an enumeration called `ASCIIControlCharacter` are defined to be of type `Character`, and are set to some of the more common ASCII control characters. `Character` values are described in Strings and Characters.

Raw values can be strings, characters, or any of the integer or floating-point number types. Each raw value must be unique within its enumeration declaration.

> NOTE
>
> Raw values are *not* the same as associated values. Raw values are set to prepopulated values when you first define the enumeration in your code, like the three ASCII codes above. The raw value for a particular enumeration case is always the same. Associated values are set when you create a new constant or variable based on one of the enumeration's cases, and can be different each time you do so.

Implicitly Assigned Raw Values

When you're working with enumerations that store integer or string raw values, you don't have to explicitly assign a raw value for each case. When you don't, Swift will automatically assign the values for you.

For instance, when integers are used for raw values, the implicit value for each case is one more than the previous case. If the first case doesn't have a value set, its value is `0`.

The enumeration below is a refinement of the earlier `Planet` enumeration, with integer raw values to represent each planet's order from the sun:

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
}
```

In the example above, `Planet.mercury` has an explicit raw value of `1`, `Planet.venus` has an implicit raw value of `2`, and so on.

When strings are used for raw values, the implicit value for each case is the text of that case's name.

The enumeration below is a refinement of the earlier `CompassPoint` enumeration, with string raw values to represent each direction's name:

```
enum CompassPoint: String {
    case north, south, east, west
}
```

In the example above, `CompassPoint.south` has an implicit raw value of `"south"`, and so on.

You access the raw value of an enumeration case with its `rawValue` property:

```
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"
```

Initializing from a Raw Value

If you define an enumeration with a raw-value type, the enumeration automatically receives an initializer that takes a value of the raw value's type (as a parameter called `rawValue`) and returns either an enumeration case or `nil`. You can use this initializer to try to create a new instance of the enumeration.

This example identifies Uranus from its raw value of `7`:

```
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and equals Planet.uranus
```

Not all possible `Int` values will find a matching planet, however. Because of this, the raw value initializer always returns an *optional* enumeration case. In the example above, `possiblePlanet` is of type `Planet?`, or "optional `Planet`."

> NOTE
>
> The raw value initializer is a failable initializer, because not every raw value will return an enumeration case. For more information, see Failable Initializers.

If you try to find a planet with a position of `11`, the optional `Planet` value returned by the raw value initializer will be `nil`:

```
let positionToFind = 11
if let somePlanet = Planet(rawValue: positionToFind) {
    switch somePlanet {
    case .earth:
        print("Mostly harmless")
    default:
        print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position \(positionToFind)")
}
// Prints "There isn't a planet at position 11"
```

This example uses optional binding to try to access a planet with a raw value of `11`. The statement `if let somePlanet = Planet(rawValue: 11)` creates an optional `Planet`, and sets `somePlanet` to the value of that optional `Planet` if it can be retrieved. In this case, it is not possible to retrieve a planet with a position of `11`, and so the `else` branch is executed instead.

Recursive Enumerations

A *recursive enumeration* is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases. You indicate that an enumeration case is recursive by writing `indirect` before it, which tells the compiler to insert the necessary layer of indirection.

For example, here is an enumeration that stores simple arithmetic expressions:

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

You can also write `indirect` before the beginning of the enumeration, to enable indirection for all of the enumeration's cases that need it:

```
indirect enum ArithmeticExpression {
    case number(Int)
    case addition(ArithmeticExpression, ArithmeticExpression)
    case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

This enumeration can store three kinds of arithmetic expressions: a plain number, the addition of two expressions, and the multiplication of two expressions. The `addition` and `multiplication` cases have associated values that are also arithmetic expressions—these associated values make it possible to nest expressions. For example, the expression `(5 + 4) * 2` has a number on the right hand side of the multiplication and another expression on the left hand side of the multiplication. Because the data is nested, the enumeration used to store the data also needs to support nesting—this means the enumeration needs to be recursive. The code below shows the `ArithmeticExpression` recursive enumeration being created for `(5 + 4) * 2`:

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum,
  ArithmeticExpression.number(2))
```

A recursive function is a straightforward way to work with data that has a recursive structure. For example, here's a function that evaluates an arithmetic expression:

```
func evaluate(_ expression: ArithmeticExpression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// Prints "18"
```

This function evaluates a plain number by simply returning the associated value. It evaluates an addition or multiplication by evaluating the expression on the left hand side, evaluating the expression on the right hand side, and then adding them or multiplying them.