10. Tuples & Enums

Tuples

A tuple is a group of zero or more values represented as one value.

For example `("John", "Smith")` holds the first and last name of a person. You can access the inner values using the dot(`.`) notation followed by the index of the value:

```
var person = ("John", "Smith")

var firstName = person.0 // John
var lastName = person.1 // Smith
```

Named elements

You can name the elements from a tuple and use those names to refer to them. An element name is an identifier followed by a colon(:).

```
var person = (firstName: "John", lastName: "Smith")

var firstName = person.firstName // John
var lastName = person.lastName // Smith
```

Creating a tuple

You can declare a tuple like any other variable or constant. To initialize it you will need a another tuple or a tuple literal. A tuple literal is a list of values separated by commas between a pair of parentheses. You can use the dot notation to change the values from a tuple if it's declared as a variable.

```
var point = (0, 0)

point.0 = 10
point.1 = 15

point // (10, 15)
```

Note: Tuple are value types. When you initialize a variable tuple with another one it will actually create a copy.

```
var origin = (x: 0, y: 0)
var point = origin
point.x = 3
point.y = 5

print(origin) // (0, 0)
print(point) // (3, 5)
```

Types

The type of a tuple is determined by the values it has. So `("tuple", 1, true)` will be of type `(String, Int, Bool)`. You can have tuples with any combination of zero or more types.

If the tuple has only one element then the type of that tuple is the type of the element. `(Int)` is the same as `Int`. This has a strange implication: in swift every variable or constant is a tuple.

```
var number = 123

print(number) // 123
print(number.0) // 123
print(number.0.0) // 123
print(number.0.0.0) // 123
print(number.0.0.0.0.0.0.0) // 123
```

Empty tuple

`()` is the empty tuple - it has no elements. It also represents the `Void` type.

Decomposing Tuples

```
var person = (firstName: "John", lastName: "Smith")

var (firstName, lastName) = person

var (onlyFirstName, _) = person
var (_, onlyLastName) = person
```

Note: the `_` means "I don't care about that value"

Multiple assignment

You can use tuples to initialize more than one variable on a single line:

Instead of:

```
var a = 1
var b = 2
var c = 3
```

you can write:

```
var (a, b, c) = (1, 2, 3)
```

Instead of:

```
a = 1
b = 2
c = 3
```

you can write:

```
(a, b, c) = (1, 2, 3)
```

And yes! One line swap:

```
(a, b) = (b, a)
```

Returning multiple values

You can return multiple values from a function if you set the result type to a tuple. Here is a simple example of a function that return the quotient and the remainder of the division of `a` by `b`.

```
func divmod(a: Int, _ b:Int) -> (Int, Int) {
    return (a / b, a % b)
}

divmod(7, 3) // (2, 1)
divmod(5, 2) // (2, 1)
divmod(12, 4) // (3, 0)
```

Or the named version:

```
func divmod(a: Int, _ b:Int) -> (quotient: Int, remainder: Int) {
    return (a / b, a % b)
}
```

```
divmod(7, 3) // (quotient: 2, remainder:1)
divmod(5, 2) // (quotient: 2, remainder:1)
divmod(12, 4) // (quotient: 3, remainder:0)
```

Enums

An enumeration is a data type consisting of a set of named values, called members.

Defining an enumeration

You can define a new enumeration using the `enum` keyword followed by it's name. The member values are introduced using the `case` keyword.

```
enum iOSDeviceType {
    case iPhone
    case iPad
    case iWatch
}

var myDevice = iOSDeviceType.iPhone
```

Dot syntax

If the type of an enumeration is known or can be inferred then you can use the dot syntax for members.

```
// in this case the type is known
var myDevice: iOSDeviceType = .iPhone

// in this case the type can be inferred
if myDevice == .iPhone {
    print("I have an iPhone!")
}
```

Associated Values

Swift enumerations can store associated values of any type, and the value type can be different for each member. For example you might want to store a device model for the iPhone and iPad (like `"mini"` for the iPad, or `"6 Plus"` for the iPhone).

```
enum iOSDeviceType {
    case iPhone(String)
    case iPad(String)
    case iWatch
}
```

You can get the associated values by using a switch statement:

```
var myDevice = iOSDeviceType.iPhone("6")

switch myDevice {
case .iPhone(let model):
    print("iPhone \(model)")
case .iPad(let model):
    print("iPad \(model)")
case .iWatch:
    print("iWatch")
default:
    print("not an iOS device")
}

// iPhone 6
```

Note: Swift does not provide equality operators automatically for enumerations with associated values. You might be tempted to use nested switch statements in order to test equality. Don't forget the tuple pattern!

```
var myDevice = iOSDeviceType.iPhone("6")
var six = iOSDeviceType.iPhone("6")
var sixPlus = iOSDeviceType.iPhone("6 Plus")

// testing equlity with == wont work
// myDevice == six
// myDevice == sixPlus

func sameDevice(firstDevice: iOSDeviceType,
        secondDevice: iOSDeviceType) -> Bool {
    switch (firstDevice, secondDevice) {
    case (.iPhone(let a), .iPhone(let b)) where a == b:
        return true
    case (.iPad(let a), .iPad(let b)) where a == b:
        return true
    case (.iWatch, .iWatch):
        return true
    default:
        return false
    }
}

print(sameDevice(myDevice, six)) // true
print(sameDevice(myDevice, sixPlus)) // false
print(sameDevice(myDevice, .iWatch)) // false
```

Raw Values

Enums can have a raw value (a primitive type - Int, String, Character, etc.) associated with each member. The raw value will be of the same type for all members and the value for each member must be unique. When integers are use they autoincrement is a value is not defined for a member.

```
enum Direction: Int {
    case Up = 1
    case Down // will have the raw value 2
    case Left // will have the raw value 3
    case Right // will have the raw value 4
}
```

You can use raw values to create a enumeration value.

```
var direction = Direction(rawValue: 4) // .Right

print(direction) // Optional((Enum Value))
```

Note: Because not all raw values have an associated member value the raw value initializer is a failable initializer. The type of `direction` is `Direction?` not `Direction`.