VALENTINE
Page No.:
Date: / /

(*) Swift can be arranged into 3 basic groups:

1. structure
2. classes
3. enumerations

→ Swift's structures and enumerations are significantly more powerful than in most languages.

(*) All three have:

1. Properties → value associated with a type.
2. Initializers → code that initializes an instance of type
3. Instance methods → function specific to a type called on instance
4. class or static methods → function called on type itself

1. Properties

→ A property is a value associated with an instance of a type.

→ For Eg, String has the property isEmpty, which is a Bool that tells you whether the string is empty.

→ Array<T> has the property count, which is the number of elements in the array as an Int.

VALENTINE
Page No.:
Date: | |

Eg: let countingUp = ["one", "two"]
let secondElement = countingUp[1]
countingUp.count

i.e
["one", "two"]
"two"
2

let emptyString = String()
emptyString.isEmpty

i.e
true.

② Initializers.

→ Initializers are responsible for preparing the contents of a new instance of a type.
→ When an initializers is finished, the instance is ready for action.
→ To create a new instance using an initializers, you use the type name followed by a pair of parentheses, if required, arguments.

→ This signature - the combination of type and arguments - corresponds to a specific initializer.

→ Some standard types have initializers that return empty literals when no arguments are supplied.

Eg: let emptyString = String()

→ Other types have default values:

Eg: let defaultNumber = Int() 0
let defaultBool = Bool() false.

→ Types can have multiple initializers.

Eg: String has an intializer that accepts an Int and creates a string based on that value.

let number = 42
let meaningOfLife = String(number) "42"

→ To create a set, you can use the Set initializer that accepts an array literal.

Eg: let availableRooms = Set([205, 411, 412])

→ Float has several initializers.
→ The parameter-less initializer return an instance of Float with the default value.
Eg: let defaultFloat = Float() 0.0

→ There is also an initializer that accepts a floating-point literal.
Eg: let floatfromLiteral = Float(3.14)

③ instance methods
→ instance method is a function that is specific to a particular type and can be called on an instance of that type.
Eg: append(_:) method.

```
var countingUp = ["one", two"]
let secondElement = countingUp[1]
countingUp.count
countingUp.append("three")
```

↓

accepts an elements of the array's type and adds it to the end of the array.

⊛ Variables and constant

→ var keywords denotes a variable.
→ so the value can be changed from its initial value.

Eg: var str = "Hello, play" — "Hello, play"
str = "Hello, Swift" — "Hello, swift"

→ let keyword denotes a contant value.
→ Here value cannot be changed.
→ In your Swift code, you should use let unless you expect the value will need to change.

Eg: var str = "Hello, play" — "Hello, play"
str = "Hello, Str" — "Hello, str"
let constStr = str — "Hello, str"

Because constStr is a constant, attempting to change its value will cause an error.

⊛ Inferring types and Specifying types.

→ If you don't specify the type that does not mean they are untyped.
→ Instead, the compiler infers their types

from the initial values.

Eg: var str = "Hello, paly"

It will assume str as String as its initial value is string.

→ If you need to specify the type if your contant or variable don't have initial value then,

Eg: var str : String
var nextYear : Int

Collection types

→ Three collections:
① arrays
② dictionaries
③ Sets.

① Arrays
→ Array is an ordered collection of elements.
→ Written as Array<T>, where T is the

type of element that the array will contain.

→ Array can contain elements of any type : standard, structure or a class.

Eg: var arrayOfInts : Array<Int>

→ Arrays are strongly typed. Once you declare an array as containing elements of, say, Int, you cannot add a String to it.

→ There is a shorthand syntax for declaring arrays.

Eg: var arraysOfInts : [Int]

② dictionary.

→ It is unordered collection of key-value pairs.

→ Value can be any of type like structure and classes.

→ Key can also be of any type but they must be unique.

→ Key must be hashable, which allows the dictionary to guarantee that the

keys are unique and to access the value for a given key more efficiently.

→ Swift dictionary are strongly typed.

→ key and value must have same type.

eg: var dictionaryOfCapital : Dictionary<String, String>

→ Shorthand syntax for declaring dictionary.

eg: var dictionaryOfCapital : [String : String]

③ Set

→ A set is similar to array in that it contains a number of elements of a certain type.

→ Sets are unordered, and member must be unique as well as hashable.

eg: var reiningNumbers : Set<Int>