Swift AP Exam Language Reference Sheet

	Instruction (Swift)	Instruction (AP)	Explanation
Assignment, Display, and Input			

	Assignment
	`let a = expression`
	`var a = expression`
	`a = expression`
	
	Examples
	`var name = "Douglas"`
	`let min = 0`
		Text:
		`a ← expression`
		
		Block:
		`a ← expression`
			Evaluates `expression` and assigns the result to the variable `a`.

	Print Statements
	`print(expression)`
	
	Examples
	`print("Hello, World!")`
	`print(min)`
		Text:
		`DISPLAY (expression)`
		
		Block:
		`DISPLAY expression`
			Displays the value of `expression`, followed by a space.

	Because Swift handles user input differently, there is no direct parallel to `input()`.	Text:
		`INPUT()`
		
		Block:
		`INPUT`
			Accepts a value from the user and returns it.

Arithmetic Operators and Numeric Procedures			

	Arithmetic Operators
	`a + b`
	`a - b`
	`a * b`
	`a / b`
		Text and Block:
		`a + b`
		`a - b`
		`a * b`
		`a / b`
			The arithmetic operators `+`, `-`, `*`, and `/` are used to perform arithmetic on `a` and `b`.
			
			For example, `3 / 2` evaluates to `1.5`.

	Modulus (Remainder Operator)
	`a % b`
	
	Example
	`17 % 4 == 1`
		Text and Block:
		`a MOD b`
			Evaluates to the remainder when `a` is divided by `b`. Assume that `a` and `b` are positive integers.
			
			For example, `17 MOD 5` evaluates to `2`.

	Random
	`Int.random(in: start...end)`
	
	Example
	`Int.random(in: 0...100)`
		Text:
		`RANDOM (a, b)`
		
		Block:
		`RANDOM a, b`
			Evaluates to a random integer from `a` to `b`, including `a` and `b`.
			
			For example, `RANDOM (1, 3)` could evaluate to `1`, `2`, or `3`.

Relational and Boolean Operators			

	Relational Operators
	`a == b`
	`a != b`
	`a > b`
	`a < b`
	`a >= b`
	`a <= b`
		Text and Block:
		`a = b`
		`a ≠ b`
		`a > b`
		`a < b`
		`a ≥ b`
		`a ≤ b`
			The relational operators `=`, `≠`, `>`, `<`, `≥`, and `≤` are used to test the relationship between two variables, expressions, or values.
			
			For example, `a = b` evaluates to `true` if `a` and `b` are equal; otherwise, it evaluates to `false`.

	Logical NOT Operator
	`!condition`
	
	Example
	`let x = 4`
	`!(x < 5) == false`
		Text:
		`NOT condition`
		
		Block:
		`NOT condition`
			Evaluates to `true` if condition is `false`; otherwise evaluates to `false`.

	Logical AND Operator
	`&&`
	
	Example
	`let x = -1`
	`let y = 1`
	`(x < 0 && y > 0) == true`
	`(x < 0 && y < 0) == false`
		Text:
		`condition1 AND condition2`
		
		Block:
		`condition1 AND condition2`
			Evaluates to `true` if both `condition1` and `condition2` are `true`; otherwise, evaluates to `false`.

	Logical OR Operator
	`||`
	
	Example
	`let x = -1`
	`let y = 1`
	`(x > 0 || y > 0) == true`
	`(x < 0 || y > 0) == true`
	`(x > 0 || y < 0) == false`
		Text:
		`condition1 OR condition2`
		
		Block:
		`condition1 OR condition2`
			Evaluates to `true` if `condition1` is `true` or if `condition2` is `true` or if both `condition1` and `condition2` are `true`; otherwise, evaluates to `false`.

Selection			

	If Statement
	`if condition {`
	`  <block of statements>`
	`}`
	
	Example
	`// prints It's too hot!`
	`let temperature = 102`
	`if temperature > 99 {`
	`  print("It's too hot!")`
	`}`
		Text:
		`IF (condition)`
		`{`
		`  <block of statements>`
		`}`
		
		Block:
		`IF condition`
		`block of statements`
			The code in `block of statements` is executed if the Boolean expression condition evaluates to `true`; no action is taken if condition evaluates to `false`.

	If-Else Statement
	`if condition {`
	`  <first block of statements>`
	`} else {`
	`  <second block of statements>`
	`}`
	
	Example
	`// prints It's perfect!`
	`let temperature = 72`
	`if temperature > 99 {`
	`  print("It's too hot!")`
	`} else {`
	`  print("It's perfect!")`
	`}`
		Text:
		`IF (condition)`
		`{`
		`  <first block of statements>`
		`}`
		`ELSE`
		`{`
		`  <second block of statements>`
		`}`
		
		`IF condition`
		`first block of statements`
		`ELSE`
		`second block of statements`
			The code in `first block of statements` is executed if the Boolean expression `condition` evaluates to `true`; otherwise, the code in `second block of statements` is executed.

Iteration			

	For Loop
	`for item in range {`
	`  <block of statements>`
	`}`
	
	Example
	`for number in 1...100 {`
	`  print(number)`
	`}`
		Text:
		`REPEAT n TIMES`
		`{`
		`  <block of statements>`
		`}`
		
		Block:
		`REPEAT n TIMES`
		`block of statements`
			The code in `block of statements` is executed `n` times.

	While Loop
	`while condition {`
	`  <block of statements>`
	`}`
	
	Note: Swift does not have a `repeat until` operation. A while loop can be used instead but the condition is the opposite of `repeat until`.
	
	Example
	`var n = 0`
	`while n < 4 {`
	`  print(n)`
	`  n += 1`
	`}`
	is the same as:
	`REPEAT UNTIL (n >= 4)`
		Text:
		`REPEAT UNTIL (condition)`
		`{`
		`  <block of statements>`
		`}`
		
		Block:
		`REPEAT UNTIL condition`
		`block of statements`
			The code in `block of statements` is repeated until the Boolean expression `condition` evaluates to `true`.

List Operations	**In Swift, lists are zero-indexed, so the first element is at `list[0]`. If a Swift list index is less than 0 or greater than the length of the list minus 1, the program terminates with an error.**	**On the AP exam for all list operations, if a list index is less than 1 or greater than the length of the list, an error message is produced and the program terminates. There is no zero index in the AP language.**	

	Accessing an Element
	`list[index]`
	
	Example
	`let fruits = ["apple", "banana", "cherry"]`
	`print(fruits[0]) // prints apple`
	`print(fruits[1]) // prints banana`
		Text:
		`list[i]`
		
		Block:
		`list i`
			Refers to the element of `list` at index `i`. The first element of `list` is at index `1`.

	Assigning a Value
	`list[i] = list[j]`
	
	Example
	`var fruits = ["apple", "banana", "cherry"]`
	`let i = 1`
	`let j = 2`
	`fruits[i] = fruits[j]`
	`// list is now "apple", "cherry", "cherry"`
		Text:
		`list[i] ← list[j]`
		
		Block:
		`list i ← list j`
			Assigns the value of `list[j]` to `list[i]`

	Assigning Multiple Values
	`list = [value1, value2, value3]`
	
	Example
	`var fruits = ["apple", "banana", "cherry"]`
	`// list is now "apple", "cherry", "cherry"`
		Text:
		`list ← [value1, value2, value3]`
		
		Block:
		`list ← value1, value2, value3`
			Assigns `value1`, `value2`, and `value3` to `list[1]`, `list[2]`, and `list[3]`, respectively.

	For-Each Loop
	`for item in list {`
	`  <block of statements>`
	`}`
	
	Example
	`let numbers = [1,2,4,5,7,9]`
	`// prints only even numbers`
	`for number in numbers {`
	`  if number % 2 == 0 {`
	`  print(number)`
	`  }`
	`}`
		Text:
		`FOR EACH item IN list`
		`{`
		`  <block of statements>`
		`}`
		
		Block:
		`FOR EACH item IN list`
		`block of statements`
			The variable `item` is assigned the value of each element of `list` sequentially, in order from the first element to the last element.
			The code in `block of statements` is executed once for each assignment of `item`.

	Inserting a Value into a List
	`list.insert(value, at:index)`
	
	Example
	`var fruits = ["apple", "banana", "cherry"]`
	`fruits.insert("grape", at: 1)`
	`// list is now "apple", "grape", "banana", "cherry"`
		Text:
		`INSERT (list, i, value)`
		
		Block:
		`INSERT list, i, value`
			Any values in `list` at indices greater than or equal to `i` are shifted to the right. The length of list is increased by `1`, and `value` is placed at index `i` in `list`.

	Appending a Value to a List
	`list.append(value)`
	
	Example
	`var fruits = ["apple", "banana", "cherry"]`
	`fruits.append("grape")`
	`// list is now "apple", "banana", "cherry", "grape"`
		Text:
		`APPEND (list, value)`
		
		Block:
		`APPEND list, value`
			The length of list is increased by `1`, and `value` is placed at the end of `list`.

	Removing a Value from a List
	`list.remove(at:index)`
	
	Example
	`var fruits = ["apple", "banana", "cherry"]`
	`fruits.remove(at: 1)`
	`// list is now "apple", "cherry"`
		Text:
		`REMOVE (list, i)`
		
		Block:
		`REMOVE list, i`
			Removes the item at index `i` in `list` and shifts to the left any values at indices greater than `i`. The length of `list` is decreased by `1`.

	Length of a List
	`list.count`
	
	Example
	`var fruits = ["apple", "banana", "cherry"]`
	`print(fruits.count)`
		Text:
		`LENGTH (list)`
		
		Block:
		`LENGTH list`
			Evaluates to the number of elements in `list`.

Procedures	In Swift, procedures are called functions. Functions associated with a type instance are called methods.		

	Functions in Swift (No Return Value)
	`func name(label: Type) {`
	`  <instructions>`
	`}`
	
	Example
	`func greet(person:String) {`
	`  print("Hello, \(person)!")`
	`}`
	`greet(person:"Douglas")`
	`// prints Hello, Douglas!`
		Text:
		`PROCEDURE name (parameter1, parameter2, ...)`
		`{`
		`  <instructions>`
		`}`
		
		Block:
		`PROCEDURE name parameter1, parameter2,...`
		`instructions`
			A procedure, `name`, takes zero or more parameters. The procedure contains programming instructions.

	Functions in Swift (Return Value)
	`func name(label: Type) -> Type {`
	`  <instructions>`
	`  return expression`
	`}`
	
	Example
	`func greet(person:String) -> String {`
	`  return "Hello, \(person)!"`
	`}`
	`let greeting = greet(person:"Douglas")`
	`print(greeting)`
	`// prints Hello, Douglas!`
		Text:
		`PROCEDURE name (parameter1, parameter2, ...)`
		`{`
		`  <instructions>`
		`  RETURN (expression)`
		`}`
		
		Block:
		`PROCEDURE name parameter1, parameter2,...`
		`instructions`
		`RETURN expression`
			A procedure, `name`, takes zero or more parameters. The procedure contains programming instructions and returns the value of `expression`. The `RETURN` statement may appear at any point inside the procedure and causes an immediate return from the procedure back to the calling program.