A Swift Tour | Documentation

Explore the features and syntax of Swift.

Tradition suggests that the first program in a new language should print the words “Hello, world!” on the screen. In Swift, this can be done in a single line:

```
print("Hello, world!")
// Prints "Hello, world!"
```

If you have written code in C or Objective-C, this syntax looks familiar to you — in Swift, this line of code is a complete program. You don’t need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don’t need a `main()` function. You also don’t need to write semicolons at the end of every statement.

This tour gives you enough information to start writing code in Swift by showing you how to accomplish a variety of programming tasks. Don’t worry if you don’t understand something — everything introduced in this tour is explained in detail in the rest of this book.

Simple Values

Use `let` to make a constant and `var` to make a variable. The value of a constant doesn’t need to be known at compile time, but you must assign it a value exactly once. This means you can use constants to name a value that you determine once but use in many places.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

A constant or variable must have the same type as the value you want to assign to it. However, you don’t always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that `myVariable` is an integer because its initial value is an integer.

If the initial value doesn't provide enough information (or if there isn't an initial value), specify the type by writing it after the variable, separated by a colon.

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

Values are never implicitly converted to another type. If you need to convert a value to a different type, explicitly make an instance of the desired type.

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

There's an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (`\`) before the parentheses. For example:

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

Use three double quotation marks (`"""`) for strings that take up multiple lines. Indentation at the start of each quoted line is removed, as long as it matches the indentation of the closing quotation marks. For example:

```
let quotation = """
        Even though there's whitespace to the left,
        the actual lines aren't indented.
            Except for this line.
        Double quotes (") can appear without being escaped.

        I still have \(apples + oranges) pieces of fruit.
        """
```

Create arrays and dictionaries using brackets (`[]`), and access their elements by writing the index or key in brackets. A comma is allowed after the last element.

```
var fruits = ["strawberries", "limes", "tangerines"]
fruits[1] = "grapes"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
 ]
occupations["Jayne"] = "Public Relations"
```

Arrays automatically grow as you add elements.

```
fruits.append("blueberries")
print(fruits)
// Prints "["strawberries", "grapes", "tangerines", "blueberries"]"
```

You also use brackets to write an empty array or dictionary. For an array, write `[]`, and for a dictionary, write `[:]`.

```
fruits = []
occupations = [:]
```

If you're assigning an empty array or dictionary to a new variable, or another place where there isn't any type information, you need to specify the type.

```
let emptyArray: [String] = []
let emptyDictionary: [String: Float] = [:]
```

Control Flow

Use `if` and `switch` to make conditionals, and use `for`-`in`, `while`, and `repeat`-`while` to make loops. Parentheses around the condition or loop variable are optional. Braces around the body are required.

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
// Prints "11"
```

In an `if` statement, the conditional must be a Boolean expression — this means that code such as `if score { ... }` is an error, not an implicit comparison to zero.

You can write `if` or `switch` after the equal sign (`=`) of an assignment or after `return` , to choose a value based on the condition.

```
let scoreDecoration = if teamScore > 10 {
    "🎉"
} else {
    ""
}
print("Score:", teamScore, scoreDecoration)
// Prints "Score: 11 🎉"
```

You can use `if` and `let` together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains `nil` to indicate that a value is missing. Write a question mark (`?`) after the type of a value to mark the value as optional.

```
var optionalString: String? = "Hello"
print(optionalString == nil)
// Prints "false"

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

If the optional value is `nil`, the conditional is `false` and the code in braces is skipped. Otherwise, the optional value is unwrapped and assigned to the constant after `let`, which makes the unwrapped value available inside the block of code.

Another way to handle optional values is to provide a default value using the `??` operator. If the optional value is missing, the default value is used instead.

```
let nickname: String? = nil
let fullName: String = "John Appleseed"
let informalGreeting = "Hi \(nickname ?? fullName)"
```

You can use a shorter spelling to unwrap a value, using the same name for that unwrapped value.

```
if let nickname {
    print("Hey, \(nickname)")
}
// Doesn't print anything, because nickname is nil.
```

Switches support any kind of data and a wide variety of comparison operations — they aren't limited to integers and tests for equality.

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    print("Add some raisins and make ants on a log.")
case "cucumber", "watercress":
    print("That would make a good tea sandwich.")
case let x where x.hasSuffix("pepper"):
    print("Is it a spicy \(x)?")
default:
    print("Everything tastes good in soup.")
}
// Prints "Is it a spicy red pepper?"
```

Notice how `let` can be used in a pattern to assign the value that matched the pattern to a constant.

After executing the code inside the switch case that matched, the program exits from the switch statement. Execution doesn't continue to the next case, so you don't need to explicitly break out of the switch at the end of each case's code.

You use `for`-`in` to iterate over items in a dictionary by providing a pair of names to use for each key-value pair. Dictionaries are an unordered collection, so their keys and values are iterated over in an arbitrary order.

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (_, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)
// Prints "25"
```

Use `while` to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```
var n = 2
while n < 100 {
    n *= 2
}
print(n)
// Prints "128"

var m = 2
repeat {
    m *= 2
} while m < 100
print(m)
// Prints "128"
```

You can keep an index in a loop by using `..<` to make a range of indexes.

```
var total = 0
for i in 0..<4 {
    total += i
}
print(total)
// Prints "6"
```

Use `..<` to make a range that omits its upper value, and use `...` to make a range that includes both values.

Functions and Closures

Use `func` to declare a function. Call a function by following its name with a list of arguments in parentheses. Use `->` to separate the parameter names and types from the function's return type.

```
func greet(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet(person: "Bob", day: "Tuesday")
```

By default, functions use their parameter names as labels for their arguments. Write a custom argument label before the parameter name, or write `_` to use no argument label.

```
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

Use a tuple to make a compound value — for example, to return multiple values from a function. The elements of a tuple can be referred to either by name or by number.

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
```

```
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}
let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics.sum)
// Prints "120"
print(statistics.2)
// Prints "120"
```

Functions can be nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that's long or complex.

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

Functions are a first-class type. This means that a function can return another function as its value.

```
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
```

```
var increment = makeIncrementer()
increment(7)
```

A function can take another function as one of its arguments.

```
func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(list: numbers, condition: lessThanTen)
```

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it's executed — you saw an example of this already with nested functions. You can write a closure without a name by surrounding code with braces (`{}`). Use `in` to separate the arguments and return type from the body.

```
numbers.map({ (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

You have several options for writing closures more concisely. When a closure's type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both. Single statement closures implicitly return the value of their only statement.

```
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
// Prints "[60, 57, 21, 36]"
```

You can refer to parameters by number instead of by name — this approach is especially useful in very short closures. A closure passed as the last argument to a function can appear immediately after the parentheses. When a closure is the only argument to a function, you can omit the parentheses entirely.

```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
// Prints "[20, 19, 12, 7]"
```

Objects and Classes

Use `class` followed by the class's name to create a class. A property declaration in a class is written the same way as a constant or variable declaration, except that it's in the context of a class. Likewise, method and function declarations are written the same way.

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

This version of the `Shape` class is missing something important: an initializer to set up the class when an instance is created. Use `init` to create one.

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
```

```
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

Notice how `self` is used to distinguish the `name` property from the `name` argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned — either in its declaration (as with `numberOfSides`) or in the initializer (as with `name`).

Use `deinit` to create a deinitializer if you need to perform some cleanup before the object is deallocated.

Subclasses include their superclass name after their class name, separated by a colon. There's no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.

Methods on a subclass that override the superclass's implementation are marked with `override` — overriding a method by accident, without `override`, is detected by the compiler as an error. The compiler also detects methods with `override` that don't actually override any method in the superclass.

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() -> Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

In addition to simple properties that are stored, properties can have a getter and a setter.

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
        get {
             return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0
        }
    }

    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \(sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)
// Prints "9.3"
triangle.perimeter = 9.9
print(triangle.sideLength)
// Prints "3.3000000000000003"
```

In the setter for `perimeter`, the new value has the implicit name `newValue`. You can provide an explicit name in parentheses after `set`.

Notice that the initializer for the `EquilateralTriangle` class has three different steps:

1. Setting the value of properties that the subclass declares.

2. Calling the superclass's initializer.

3. Changing the value of properties defined by the superclass. Any additional setup work that uses methods, getters, or setters can also be done at this point.

If you don't need to compute the property but still need to provide code that's run before and after setting a new value, use `willSet` and `didSet`. The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is always the same as the side length of its square.

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength = newValue.sideLength
        }
    }
    var square: Square {
        willSet {
            triangle.sideLength = newValue.sideLength
        }
    }
    init(size: Double, name: String) {
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength: size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shap
e")
print(triangleAndSquare.square.sideLength)
// Prints "10.0"
print(triangleAndSquare.triangle.sideLength)
// Prints "10.0"
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
print(triangleAndSquare.triangle.sideLength)
// Prints "50.0"
```

When working with optional values, you can write `?` before operations like methods, properties, and subscripting. If the value before the `?` is `nil`, everything after the `?` is ignored and the value of the whole expression is `nil`. Otherwise, the optional value is unwrapped, and everything after the `?` acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.

```
let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional squar
e")
let sideLength = optionalSquare?.sideLength
```

Enumerations and Structures

Use `enum` to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```
enum Rank: Int {
    case ace = 1
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king

    func simpleDescription() -> String {
        switch self {
        case .ace:
            return "ace"
        case .jack:
            return "jack"
        case .queen:
            return "queen"
        case .king:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
}
let ace = Rank.ace
let aceRawValue = ace.rawValue
```

By default, Swift assigns the raw values starting at zero and incrementing by one each time, but you can change this behavior by explicitly specifying values. In the example above, `Ace` is explicitly given a raw value of `1`, and the rest of the raw values are assigned in order. You can also use strings or floating-point numbers as the raw type of an enumeration. Use the `rawValue` property to access the raw value of an enumeration case.

Use the `init?(rawValue:)` initializer to make an instance of an enumeration from a raw value. It returns either the enumeration case matching the raw value or `nil` if there's no matching `Rank`.

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

The case values of an enumeration are actual values, not just another way of writing their raw values. In fact, in cases where there isn't a meaningful raw value, you don't have to provide one.

```
enum Suit {
    case spades, hearts, diamonds, clubs

    func simpleDescription() -> String {
        switch self {
        case .spades:
            return "spades"
        case .hearts:
            return "hearts"
        case .diamonds:
            return "diamonds"
        case .clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()
```

Notice the two ways that the `hearts` case of the enumeration is referred to above: When assigning a value to the `hearts` constant, the enumeration case `Suit.hearts` is referred to by its full name because the constant doesn't have an explicit type specified. Inside the switch, the enumeration case is referred to by the abbreviated form `.hearts` because the value of `self` is already known to be a suit. You can use the abbreviated form anytime the value's type is already known.

If an enumeration has raw values, those values are determined as part of the declaration, which means every instance of a particular enumeration case always has the same raw value. Another choice for enumeration cases is to have values associated with the case — these values are determined when you make the instance, and they can be different for each instance of an enumeration case. You can think of the associated values as behaving like stored properties of the enumeration case instance. For example, consider the case of requesting the sunrise and sunset times from a server. The server either responds with the requested information, or it responds with a description of what went wrong.

```
enum ServerResponse {
    case result(String, String)
    case failure(String)
}
```

```
let success = ServerResponse.result("6:00 am", "8:09 pm")
let failure = ServerResponse.failure("Out of cheese.")

switch success {
case let .result(sunrise, sunset):
    print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
case let .failure(message):
    print("Failure...  \(message)")
}
// Prints "Sunrise is at 6:00 am and sunset is at 8:09 pm."
```

Notice how the sunrise and sunset times are extracted from the `ServerResponse` value as part of matching the value against the switch cases.

Use `struct` to create a structure. Structures support many of the same behaviors as classes, including methods and initializers. One of the most important differences between structures and classes is that structures are always copied when they're passed around in your code, but classes are passed by reference.

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription
())"
    }
}
let threeOfSpades = Card(rank: .three, suit: .spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

Concurrency

Use `async` to mark a function that runs asynchronously.

```
func fetchUserID(from server: String) async -> Int {
    if server == "primary" {
        return 97
    }
    return 501
}
```

You mark a call to an asynchronous function by writing `await` in front of it.

```
func fetchUsername(from server: String) async -> String {
    let userID = await fetchUserID(from: server)
    if userID == 501 {
        return "John Appleseed"
    }
    return "Guest"
}
```

Use `async let` to call an asynchronous function, letting it run in parallel with other asynchronous code. When you use the value it returns, write `await`.

```
func connectUser(to server: String) async {
    async let userID = fetchUserID(from: server)
    async let username = fetchUsername(from: server)
    let greeting = await "Hello \(username), user ID \(userID)"
    print(greeting)
}
```

Use `Task` to call asynchronous functions from synchronous code, without waiting for them to return.

```
Task {
    await connectUser(to: "primary")
}
// Prints "Hello Guest, user ID 97"
```

Protocols and Extensions

Use `protocol` to declare a protocol.

```
protocol ExampleProtocol {
     var simpleDescription: String { get }
     mutating func adjust()
}
```

Classes, enumerations, and structures can all adopt protocols.

```
class SimpleClass: ExampleProtocol {
     var simpleDescription: String = "A very simple class."
     var anotherProperty: Int = 69105
     func adjust() {
          simpleDescription += "  Now 100% adjusted."
     }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
     var simpleDescription: String = "A simple structure"
     mutating func adjust() {
          simpleDescription += " (adjusted)"
     }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

Notice the use of the `mutating` keyword in the declaration of `SimpleStructure` to mark a method that modifies the structure. The declaration of `SimpleClass` doesn't need any of its methods marked as mutating because methods on a class can always modify the class.

Use `extension` to add functionality to an existing type, such as new methods and computed properties. You can use an extension to add protocol conformance to a type that's declared elsewhere, or even to a type that you imported from a library or framework.

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
 }
print(7.simpleDescription)
// Prints "The number 7"
```

You can use a protocol name just like any other named type — for example, to create a collection of objects that have different types but that all conform to a single protocol. When you work with values whose type is a boxed protocol type, methods outside the protocol definition aren't available.

```
let protocolValue: any ExampleProtocol = a
print(protocolValue.simpleDescription)
// Prints "A very simple class.  Now 100% adjusted."
// print(protocolValue.anotherProperty)  // Uncomment to see the error
```

Even though the variable `protocolValue` has a runtime type of `SimpleClass`, the compiler treats it as the given type of `ExampleProtocol`. This means that you can't accidentally access methods or properties that the class implements in addition to its protocol conformance.

Error Handling

You represent errors using any type that adopts the `Error` protocol.

```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

Use `throw` to throw an error and `throws` to mark a function that can throw an error. If you throw an error in a function, the function returns immediately and the code that called the function handles the error.

```
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

There are several ways to handle errors. One way is to use `do`-`catch`. Inside the `do` block, you mark code that can throw an error by writing `try` in front of it. Inside the `catch` block, the error is automatically given the name `error` unless you give it a different name.

```
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
} catch {
    print(error)
}
// Prints "Job sent"
```

You can provide multiple `catch` blocks that handle specific errors. You write a pattern after `catch` just as you do after `case` in a switch.

```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
// Prints "Job sent"
```

Another way to handle errors is to use `try?` to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is `nil`. Otherwise, the result is an optional containing the value that the function returned.

```
let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")
let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")
```

Use `defer` to write a block of code that's executed after all other code in the function, just before the function returns. The code is executed regardless of whether the function throws an error. You can use `defer` to write setup and cleanup code next to each other, even though they need to be executed at different times.

```swift
var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]

func fridgeContains(_ food: String) -> Bool {
    fridgeIsOpen = true
    defer {
        fridgeIsOpen = false
    }

    let result = fridgeContent.contains(food)
    return result
}
fridgeContains("banana")
print(fridgeIsOpen)
// Prints "false"
```

Generics

Write a name inside angle brackets to make a generic function or type.

```swift
func makeArray<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {
    var result: [Item] = []
    for _ in 0..<numberOfTimes {
         result.append(item)
    }
    return result
}
makeArray(repeating: "knock", numberOfTimes: 4)
```

You can make generic forms of functions and methods, as well as classes, enumerations, and structures.

```swift
// Reimplement the Swift standard library's optional type
enum OptionalValue<Wrapped> {
    case none
    case some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .none
possibleInteger = .some(100)
```

Use `where` right before the body to specify a list of requirements — for example, to require the type to implement a protocol, to require two types to be the same, or to require a class to have a particular superclass.

```
func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool
    where T.Element: Equatable, T.Element == U.Element
{
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
   return false
}
anyCommonElements([1, 2, 3], [3])
```

Writing `<T: Equatable>` is the same as writing `<T> ... where T: Equatable`.

- A Swift Tour
- Simple Values
- Control Flow
- Functions and Closures
- Objects and Classes
- Enumerations and Structures
- Concurrency
- Protocols and Extensions
- Error Handling
- Generics