

PIOTR FULMAŃSKI

Learn Swift by examples

BEGINNER LEVEL

EARLY ACCESS VERSION

EDITION 1.0, JUNE 2021



SIMPLE INTRODUCTION SERIES

Learn Swift by examples

Beginner level

SIMPLE INTRODUCTION SERIES

Copyright © 2021-2022, Piotr Fulmański

All rights reserved

www: <https://fulmanski.pl>

email: book@fulmanski.pl

GitHub: <https://github.com/fulmanp/Learn-Swift-by-examples-beginner>

Edition: 1

First published: 1.0, January 2022 (planned)

This edition: 1.0, June 2021 (early access)

Build number: 202106251400

ISBN 978-83-957405-1-0



eBook (pdf, epub)

ISBN-13: 978-83-957405-1-0



While the author has used good faith efforts to ensure that the information and instructions contained in this work are accurate, the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. The author makes no warranty, express or implied, with respect to the material contained herein.

If any code samples, software or any other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Release Notes

(**A** – add, **N** – new, **U** – update)

Edition 1.0

release: June 2021

- List of topics and terms (**N**)
- Chapter 2: Variables and functions (**A**) [add escaping closures]
- Chapter 9: Protocols and generics (**A**) [add more examples]
- Chapter 10: Miscellaneous topics (**A**) [add case-let pattern]

Edition 1.0

release: May 2021

- Chapter 7: Code completion (**A**) [add solutions to tasks]
- Chapter 9: Protocols and generics (**A**) [add image, add new subsections]

Edition 1.0

release: April 2021

- Chapter 1, Section 1: Battleship game (**N**)
- Chapter 6: Properties, dictionaries and sets (**N**)
- Chapter 7: Code completion (**N**)
- Chapter 8: Structures, inheritance and errors handling (**N**)
- Chapter 9: Protocols and generics (**N**)
- Chapter 10: Miscellaneous topics (**N**)

Edition 1.0

release: March 2021

- Preface (**N**)
- Chapter 1: Initial steps (**N**)
- Chapter 2: Variables and functions (**N**)
- Chapter 3: Arrays and enumerations (**N**)
- Chapter 4: Type methods, guards and string interpolations (**N**)
- Chapter 5: Tuples, switch and extensions (**N**)

Table of contents

Preface	xv
Initial steps	23
Battleship game.....	25
Create project.....	29
Variables and functions	37
Variables.....	39
Optionals.....	49
Functions.....	57
More about functions – closures.....	71
Arrays and enumerations	87
Arrays.....	89
Enumerations	101
Range operators	107
Game code – add initializer	109
Type methods, guards and string interpolation.....	115
Type methods.....	117
Guards	123
String interpolation.....	129
Game code	133
Tuples, switch and extensions	137
Tuples	139
switch - case statement	143
Extensions	149
Game code.....	157

Properties, dictionaries and sets.....	165
Property types.....	167
Dictionaries.....	179
Sets.....	185
Game code	189
Code completion	201
Tasks to complete, part 1	203
Tests for part 1	209
Tasks solutions, part 1	215
Tasks to complete, part 2	221
Tasks solutions, part 2	229
Structures, inheritance and errors handling.....	233
Structures	235
Inheritance.....	243
Type checking and casting.....	249
Access control	259
Errors handling.....	265
Protocols and generics	271
Protocols.....	273
Structs and inheritance.....	295
Generics	305
Miscellaneous topics	317
This is not the end	319
case-let pattern	321
Bibliography.....	327

List of topics and terms

array

- predefined size
- concatenate
- range
- iterate over

case

- bind values
- `case-let`
- `for-case-let`
- `guard-case-let`
- `if-case-let`
- pattern matching
- `switch-case`

class

- access control
- class method (type method)
- `final`
- function (method, action)
- instance
- instance method
- object

closure

- trailing closure
- escaping closure (`@escaping`)

constant

`defer` (see also: error)

dictionary

enumeration (`enum`)

- enumeration associated values

error

- `throw`
- `do-catch`

extension

fall through

force unwrap (`!`)

function (see also: method)

- anonymous function
- argument label
- default parameter
- in-out parameter
- nested function
- parameter name
- return implicit
- throwing function
- type
- unwanted result (`_`)
- variadic parameter

generic

`guard`

inheritance

initializer

instance method 114

method (action)

- class (type) method
- instance method
- mutating method
- `override`

operator

- closed range operator (`...`)
- half-open range operator (`..<`)

optional

- binding
- implicitly unwrapped optionals (`!`)
- multiple optional chaining
- nil-coalescing operator (`??`)

polymorphism

property (field)

- computed property

- lazy stored property
- property observer
- stored property
- type property
- property wrapper

protocol

- associated type
- protocol composition
- protocol default implementation
- **required**

string interpolation

structure

switch-case

- fall through
- see also: **case**

tuple

type

- type annotation
- associated type
- reference type
- value type

variable

Preface

Who this book is for

This book is addressed to **all the people who want to learn basics of Swift**. Although I pay great attention to make all the contents intuitive and easy to follow, **this is not a book to learn programming** – I assume you have at least basic programming skills. Moreover, you should have basic general programming knowledge (what a loop is, what kinds of loops we have, what a conditional statement is etc) as well as basic knowledge about object oriented programming paradigms (what a class is, methods and properties). **This book is NOT intended to give you a complete Swift programming overview**. I will not discuss all the topic exhaustively; of course I do my best to show as much aspect of Swift as I only can but not in price of clearness, readability and understandability. **I want to give you as many examples as I only can. My idea is to give all the people who ever known any other (imperative) programming language a good starting point to work with and learn Swift**. If you think: *Great, I know C (or Java, Python, Pascal, etc) and I want to know some basics of Swift to get know if it's worth to spend my time to learn it* – then this book is for you.

Early access

This book is a work in progress, presented in early access version. Early access allows to publish and share some ideas before the final version appears. This way, participating in an early access, you may contribute

how the final version will look like. English is not my native language and I know that I make a lot of mistakes but I hope that text is more than readable and at least a little bit understandable. I believe that everything can be better and there is always a space for improvements. I can say that **the Toyota Way is the way I live and work focusing on continuous improvement, and respect for people**. That is why I would be very grateful if you somehow contribute improving this book. Any comments, corrections and suggestions are more than welcome. I write this book not for myself but to share what I know with others, so help me make it's contents better.

I want this book to be available for free in full version. However **if you want to financially support me you can choose a paid option**. I need money to finish it and to make it better. Everything costs. The most precious is time. I, as all other people, have to work to live and to support my family. And this consumes most of my days. Then I have a choice: play with kids or write a book. I choose playing with kids. I don't want to reduce a time spent with my family because I don't want to be a virtual parent. So I have just a little time for book. Paying for a book you allow me to reduce my job engagement (from full-time to half-time) and spending more time on book without sacrificing my family life. Moreover, having money I can pay for professional translation, text correction or simply buy better images.

I believe there is no book like this on the market and I want to make it better and better. I can do this. I don't have to ask publisher if they agree to prepare another version. If something deserves for improvement I simply do this and publish right after that.

What will you learn in this book?

I don't want to write this book and forget. My idea is to keep it as accurate and up to date as it is only possible so you can expect updates in a future even if I reach stable "final" version. As I wrote above, always

there is something to improve. As for now book covers the following topics:

Preface This is what you are reading right now. Here I explain what you can expect in this book. I also try to convince you to actively participate in shaping it's contents.

Chapter 1 Initial steps. Battleship game is a simple game but there are few variants so I will present rules you have to preserve during implementation. Next you will create a project. In this chapter you will learn how to create a project.

Chapter 2 Variables and functions. You will learn about the most fundamental bricks of every code: *constants*, *variables* and *functions*. You will also get knowledge about *optional types* and create your first *class "stub"*.

Chapter 3 Arrays and enumerations. In this chapter you will create a very basic classes with properties, initializers and methods. One of them you will use to represent game board and another one to be an entry point to the whole game logic. You will learn how to create and use *arrays*. You will also get knowledge about *enumerations* and very useful *range operators*.

Chapter 4 Type methods, guards and string interpolation. In this part you will start implement method to printing game board. You will learn what *type methods* are and how they differ from *instance methods*. I will try to convince you that *guards* are not only just a replacement to *ifs*. You will also get knowledge how to use *string interpolation* to construct a new string from a mix of constants, variables, literals, and expressions by including their values inside a string literal.

Chapter 5 Tuples, switch and extensions. You will finish implementing printing game board method. You will also create a class related to ships and implement one method to use with this type. You will learn about

tuples – "small thing" making your programmers life simpler. I will show you how Swift turns switch-case statement into handy tool and how to separate your code with extensions.

Chapter 6 Properties, dictionaries and sets. You will finish implementing printing game board method. You will also create a class related to ships and implement one method to use with this type. You will learn about various property types. After reading this chapter you will have also good understanding how to create and use two important data structures: *dictionaries* and *sets*.

Chapter 7 Code completion. This chapter allows you to test yourself. I will give you some simple tasks related to battleship game and you will try to implement adequate solutions. Next you can compare your code with mine. In this chapter you will finish game code.

Chapter 8 Structures, inheritance and errors handling. In this part you will extend your knowledge with information on more advanced topics. You will learn about *structures*, *inheritance* and *exceptions*.

Chapter 9 Protocols and generics. In this part you will continue extending your knowledge with information on more advanced topics. You will learn about protocols and writing write general purpose code with generics.

Chapter 10 This is not the end. This chapter is a summary chapter.

What you will NOT learn in this book?

This book is not intended to give a complete Swift programming overview. I will not discuss all the topic exhaustively, and only deal with topics that are necessary to understand the content and most important features of Swift.

Final word

If some part of text or code is not clear and you have problems to understand it, please let me know. I will try to rewrite it to make it more clear. Remember: **this book is written by me for you not just for my complacency.**

Give this book a try, and please let me know what you think. Any feedback is very much encouraged and welcomed! If you think that my time is worth this effort, you can support what I'm doing now and help me finalize this project. Please use email (book@fulmanski.pl) or GitHub (<https://github.com/fulmanp/Learn-Swift-by-examples-beginner/issues>) to give your positive or negative, but in all cases constructive, feedback.

Thank you for your engagement.

Piotr Fulmański

Conventions used in this book

For your convenience I will use the following typographical convention:

Italic

Indicates new terms.

Italic

Indicates old terms but for some reason I want to distinguish them from normal text flow, definitions, citations.

`Constant width`

Indicates filenames, file extensions, text of Xcode messages.

`Constant width`

Indicates commands or any other text that you should type literally (as it is given), for example text filed input.

`Constant width`

Indicates anything which is related to source code but is placed inline.

Bold

Indicates application names, menu. It also indicates statements which you need to pay special attention to.

Sometimes it is used in combination with previous styles, for example:

`Constant width with bolded part`

This way I will mark for example crucial parameter in some important command.

This is how source
code is displayed

This is how a terminal
text is displayed with
bolded command prompt.

Something worth to remember or just one-sentence summary of
some part of a section or chapter.

NOTE

Note block

I use this block to give you some additional explanation or information,
possibly loosely related to a main text.

Initial steps

You will do:

Battleship game is a simple game but there are few variants so I will present rules you have to preserve during implementation. Next you will create a project.

You will learn:

- You will learn how to create a project.

Battleship game

Showing you all Swift's concept I try to wrap them in one consistent form. I decide to build my narration around simple game – simple enough not to have to explain complex rules and at the same time giving enough flexibility to be able to show as much as possible.

Battleship is known worldwide as a pencil and paper game which dates from World War I. The game is played on four grids, two for each player. On one grid the player arranges ships and records the shots by the opponent. On the other grid the player records their own shots.

The grids are typically square – usually 10 by 10 – and the individual cells (squares) in the grid are identified by letter and number (one of them to specify row and the other for column). In version you will implement, numbers will be used for both rows and columns – you will provide cell coordinates as a pair of numbers, where first component corresponds to row and second to column. This assumption allows to create arbitrary large game boards (grids) without fear of running out of letters to indicate one of coordinates. Moreover you can treat both coordinates as integer values and use directly to index array element.

Before play begins, each player secretly arranges their ships on their primary grid. Each ship occupies a number of consecutive cells on the grid, arranged either horizontally or vertically. The number of cells for each ship is determined by the type of the ship. The ships cannot overlap (i.e., only one ship can occupy any given cell in the grid) or touch themselves (there must be minimum one cell separating two cells

belonging to two different ships). The types and numbers of ships allowed are the same for each player. These may vary depending on the rules. In version you will implement, you may use any number of ships of any size you want of course, as long as they fit on the board without breaking any game rules. Below I give some examples of correct and incorrect ships arrangements (1 ship of size 4, 2 of size 3, 3 of size 2 and 4 of size 1):

Correct	Incorrect
<pre> 1 1234567890 1..XX.....X 2..... 3..XX..X.... 4.....X.X.. 5...X.X.... 6...X.X.X.. 7...X..... 8..... 9XX....XXX. 10...X..... </pre>	<pre> 1 1234567890 1D.....X.. 2D..... 3.EEE..... 4.....AA..X 5.....A.... 6.....A.... 7.X..... 8.....B&.. 9F.....C.. 10FG.....C.. </pre>

To allow easier identification of ships breaking rules, I marked them with letters different than X:

- Ship **A**: this is a ship of size 4 and it doesn't lay in a straight line.
- Ship **B** is of size 3 and overlaps (cell marked with & character) with ship **C** of size 3.
- Ship **D** is of size 2 and touches diagonally ship **E** of size 3.
- Ship **F** is of size 2 and touches linearly ship **G** of size 1.

After the ships have been positioned, the game proceeds in a series of rounds. In each round, each player takes a turn to announce a target cell in the opponent's grid which is to be shot at. The opponent replies whether or not the cell is occupied by a ship and in consequence whether it is a *hit* or *miss*. Both players mark result (hit or miss) on their

respective grids. The attacked player do this to know how many ships are still under his or her command. The attacking player marks shot results, in order to build up a picture of the opponent's fleet.

When all of the cells of a ship have been hit, the ship's owner announces its sinking. If all of a player's ships have been sunk, the game is over and their opponent wins.

Create project

Creating a project is quite simple and involves just a few steps. To play with Swift you have to use right tool. If you have an Apple computer, Xcode application is a right choice.

On Apple computer please follow these steps:

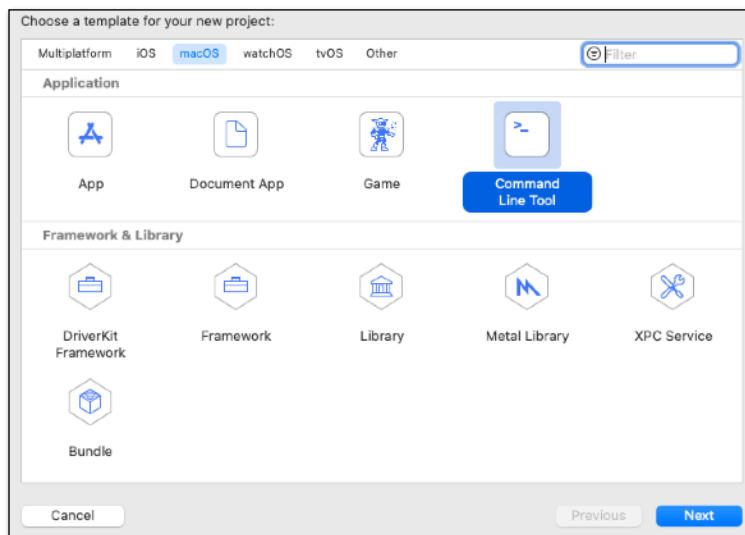
1. Find **Xcode** and run it clicking on it's icon:



2. Just after start a welcome screen will be displayed:

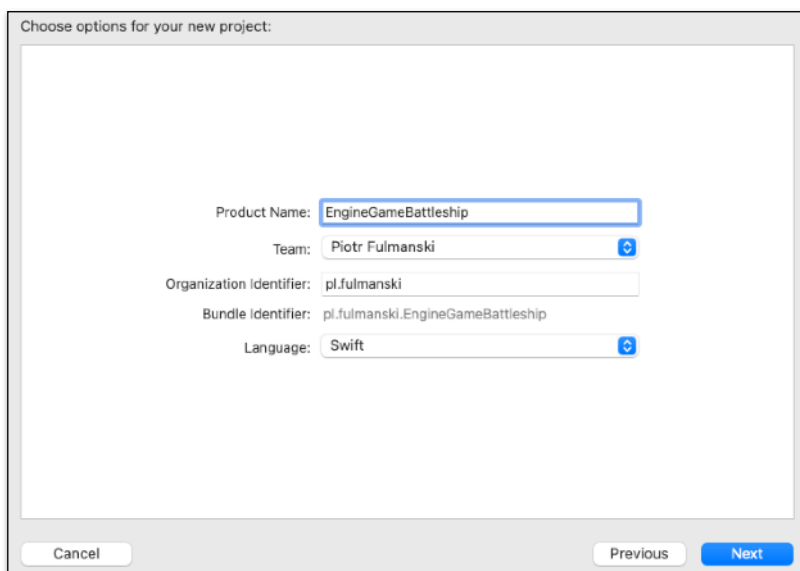


Select **Create a new Xcode project**. If this window is not present on a screen press **Shift + Command + N** or select from top menu **File / New / Project....** In either case a new project template selection window is displayed:



Select **Command line tool**.

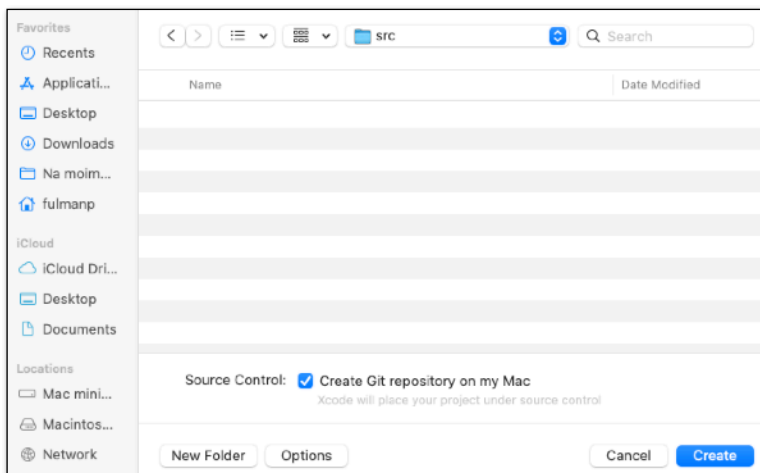
3. Next you have to provide some project identification data:



- **Product Name** – it is a name of our "product". Choose it carefully, because changing it in a future, although possible, is not so easy.
- **Team** – select your team if you have any.
- **Organization Identifier** – an identifier of an "organization"; may be a company name, developer name, reverse domain name as it is in my case or any other name.
- **Bundle Identifier** – this is something which identifies your "product" and is formed as a result of concatenating **Organization Identifier** and **Product Name**. Must be unique if you want one day upload your product to **App Store**. This is why **Organization Identifier** should be something really unique – reverse domain name is one of a method to meet this condition. As long as we only test some code or we don't want to put it in the **App Store** both names may be any strings you want.
- **Language** – select **Swift**.

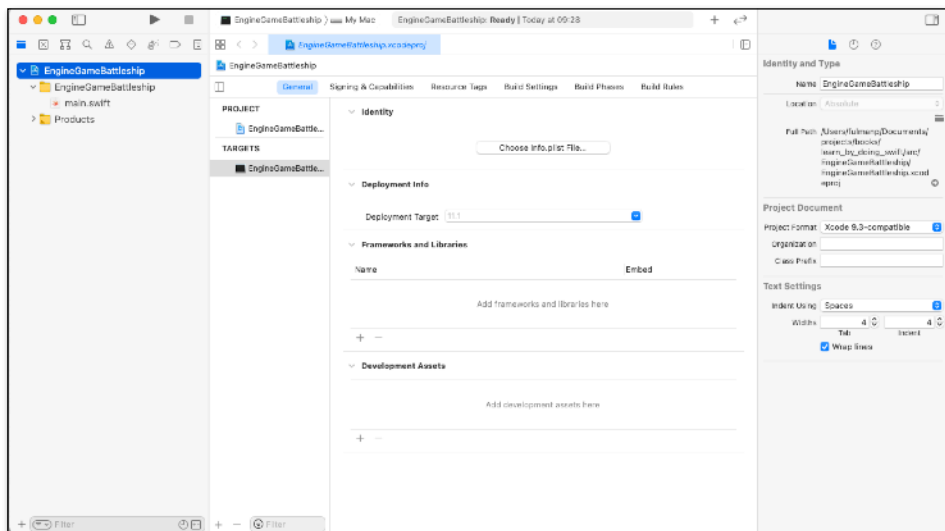
Click **Next** button.

4. Now you have to select location where you want to save your project:

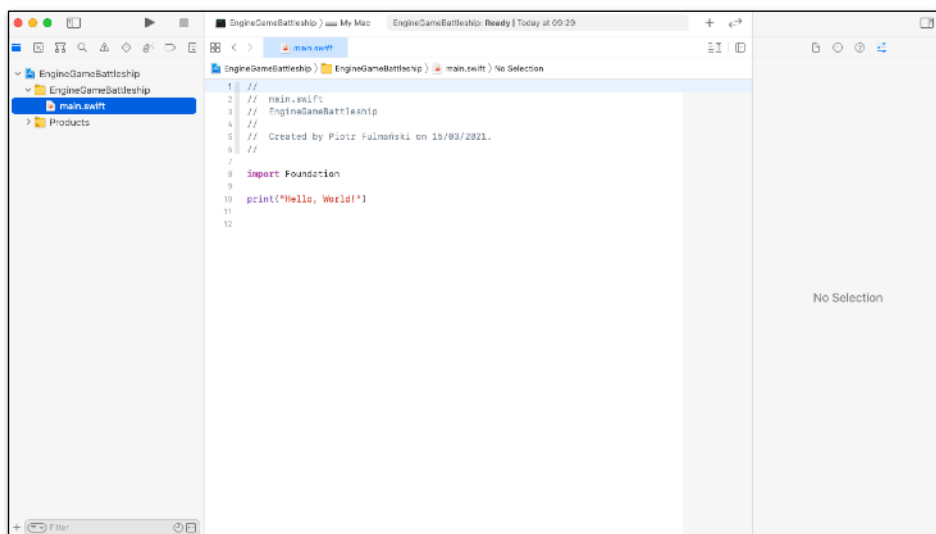


If you want to keep the code under source control, check **Create Git repository on my Mac**.

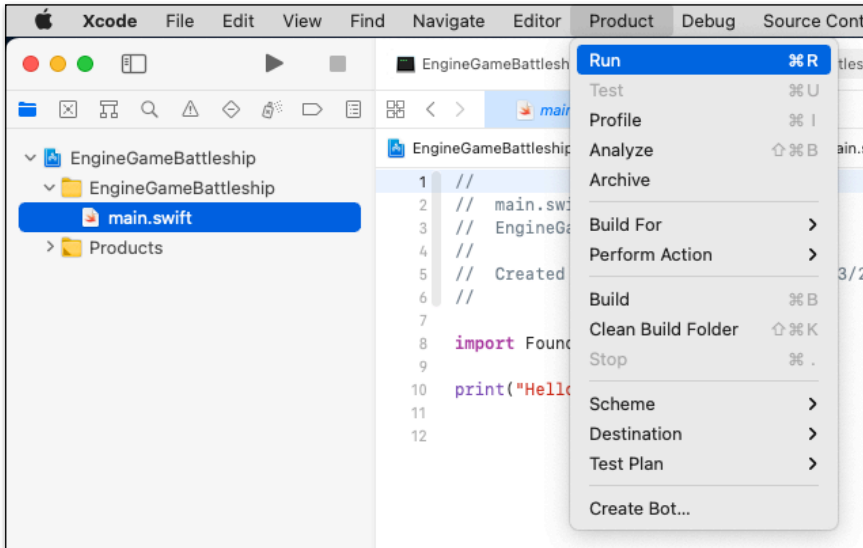
This was the last step and main Xcode window will appear:



Selecting `main.swift` file from the tree displayed in the **Project navigator** located on the left part of the main window displays the code located in this file:



Press **Command + R** or select from top menu **Product / Run**, or press "play" button to compile and execute a code.

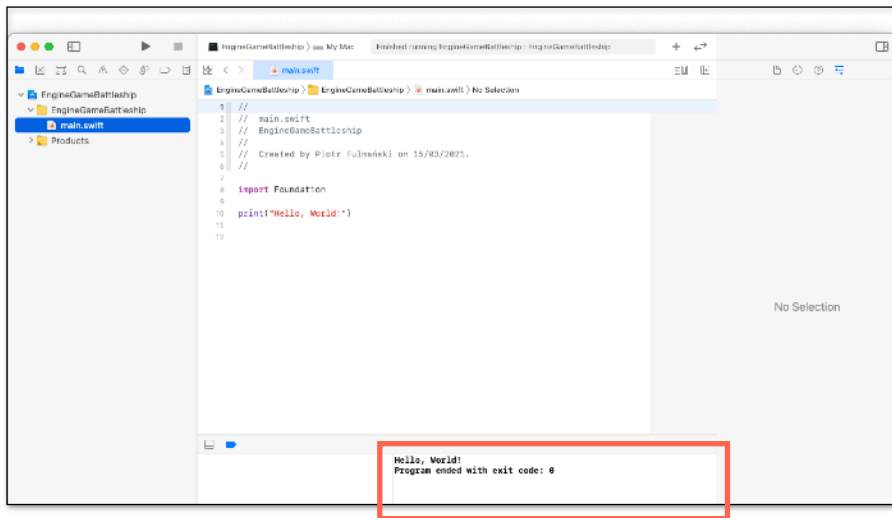


Sometimes first run takes more time than you expect. Please be patient and be sure that **Console Output** window is displayed in **Debug Area**. If not, it can be turned on with a button located in the right-bottom corner of a main window. It's the last icon on the lower right side of the panel:



Sometimes it might happen that there is no **Debug Area** window. In such a case press **Shift + Command + Y** to show or hide it. You need **Console Output** window as it is a place where all messages from your application will be displayed. Now you will see:

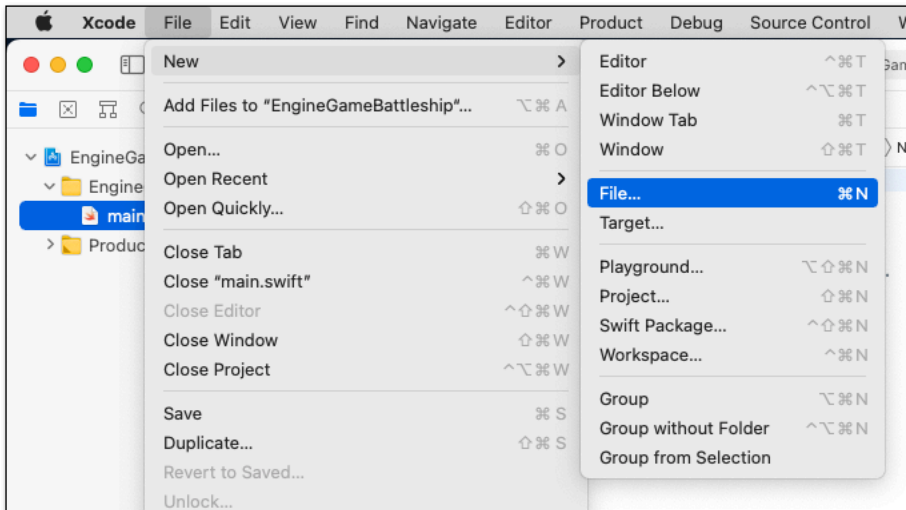
Hello, World!
Program ended with exit code: 0



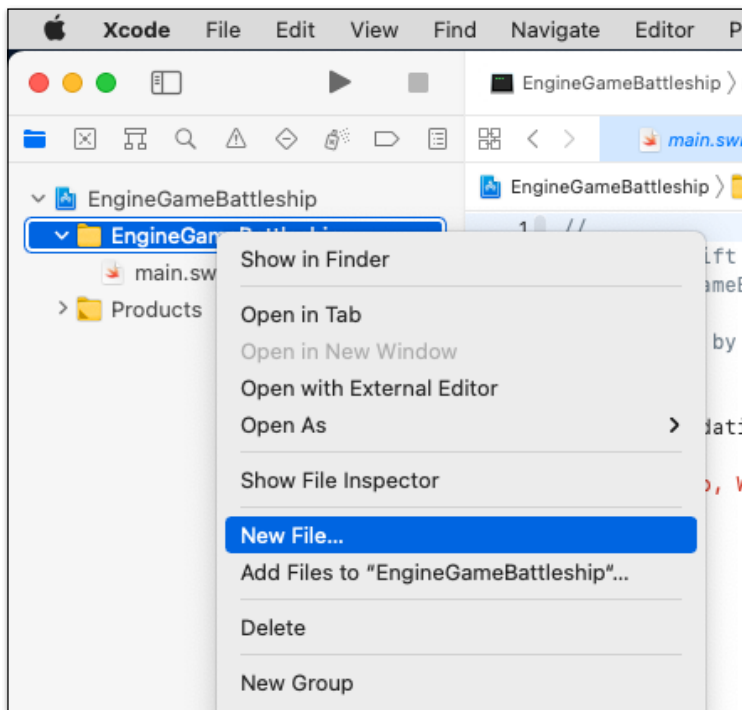
That's all. Now your project is ready and you can work on it.

Variables

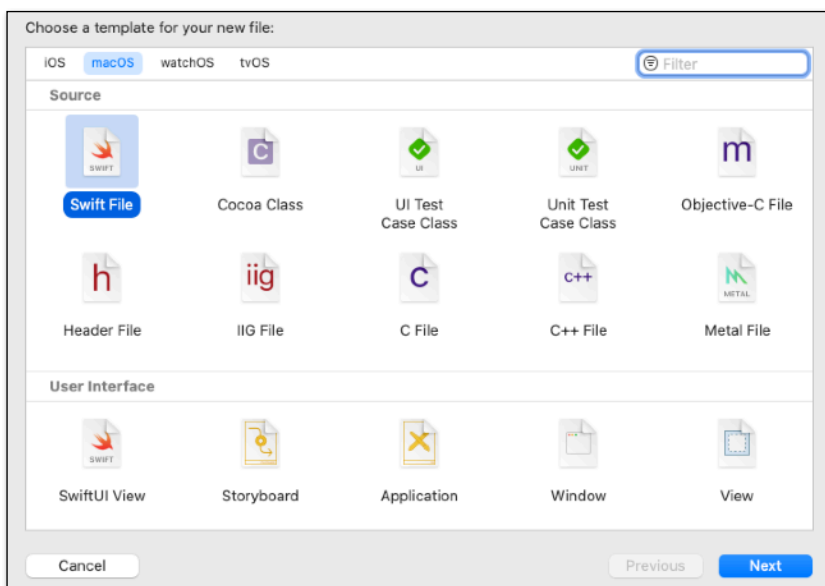
You start with creating a **Board** class to be used to represent game board in your game. Press **Command + N** or select from top menu **File / New / File...**:



Alternatively you can select **EngineGameBattleship** group in the **Project navigator**, right click on it and select from pop-up menu **New File...**:

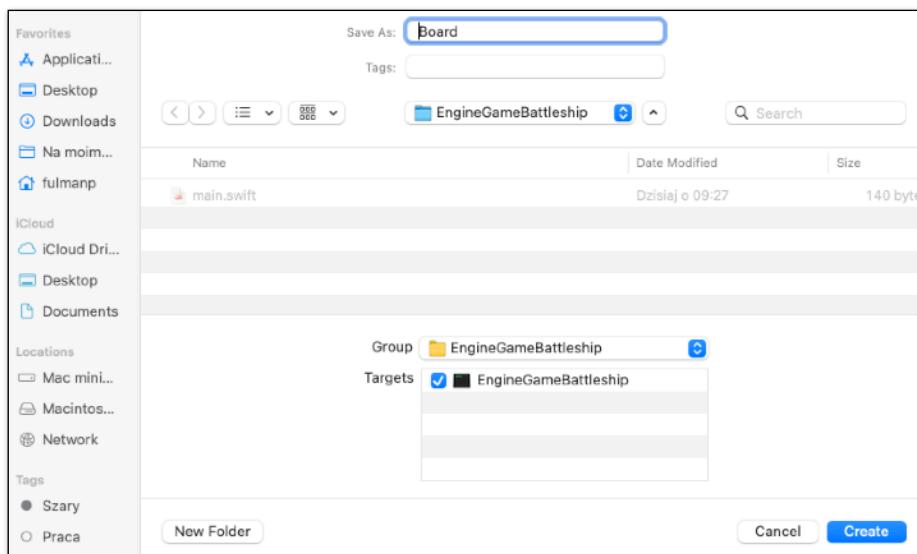


In either case you will see template selecting dialog:



Then select **macOS** and **Swift File** and accept it pressing **Next** button.

Provide file name (**Board**), which in case of Swift doesn't have to be consistent with the class name you are going to put inside (although it's a good habit):



Press **Create** to create the file. Initial contents of this file is almost empty except *import declaration*:

```
import Foundation
```

Import lets you have an access to symbols that are declared in another *module/library/framework* (different names are used). The **Foundation** is one of the fundamental framework. It provides a base layer of functionality for apps and other frameworks, including data storage and persistence, text processing, date and time calculations, sorting and filtering, and networking.

Add to `Board.swift` file your board class stub:

```
class Board {  
}
```

This is a "frame" enclosing code of a *class*. At the very first beginning add some variables to store information about board: number of *rows* and *columns* as well as *board cell types* (variables present inside a class are called *properties*):

```
class Board {  
    private let rows, cols: Int  
    private var board: [[CellType]]  
}
```

Classes (and *structures*) are general-purpose, fundamental building blocks of any programming language supporting object-oriented paradigm. With classes you define "template" to create objects (instantiate class) possessing the same set of *properties* and with the same set of *actions* (functions; sometimes known as *methods*) typical for a given type of objects. You define properties and functions in your classes and structures using the same syntax you use to define typical unrelated constants, variables, and functions.

It's worth to note that in Swift the term *instance* is preferred over the term *object*, mostly because it is right both in case of instances of classes, typically in other programming languages named an *objects*, and in case of instances of structures.

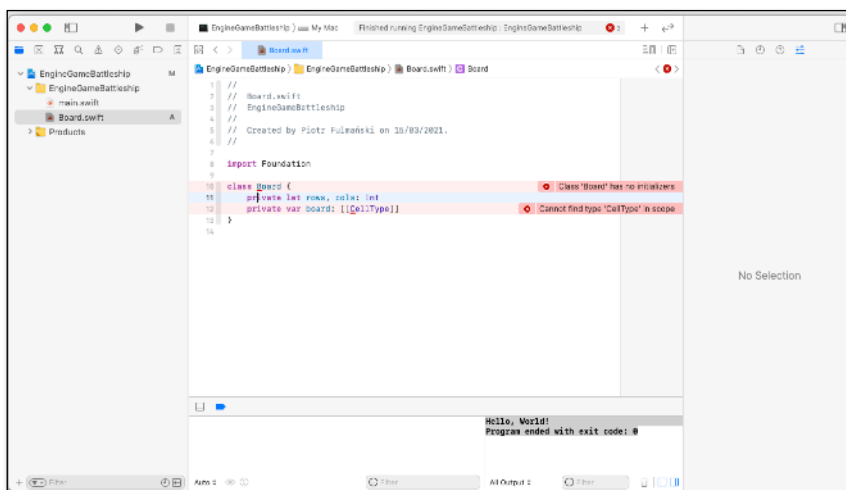
To get an access to properties or functions of instantiated class you use a typical "dot notation":

```
let instance = someClass()  
print(instance.x) // Get an access to a property 'x'  
instance.someFunction() // Call function 'someFunction'
```

See [Chapter 8: Structures, inheritance and errors handling, Section 1: Structures](#) for informations about structures and brief comparison between classes and structures.

This two simple "additions" defining three properties: **rows**, **cols** and **board**, require a lot of explanations to understand what you are doing and to fix errors which you can see know:

- Class 'Board' has no initializers
- Cannot find type 'CellType' in scope



According to Xcode's errors, there are two problems you have to solve. First error is related to properties (variables) you have in **Board** class – Swift doesn't have all the information it needs to correctly instantiate them. To fix this you have to either provide explicit values for your variables or provide a special method called *initializer* (known as constructor in other programming languages) with code initializing variables. Second error is related to unknown **CellType** type. To fix both errors you have to:

- Get knowledge about variables in Swift – you will do it in the following subsection.
- Get knowledge about functions to be able to implement initializer (initializer is a special type of function) – you will do it in a [Section 3: Functions](#) of this chapter.
- Get knowledge about arrays – you will do it in [Chapter 3: Arrays and enumerations, Section 1: Arrays](#).
- Get knowledge about enumeration types to implement `CellType` type – you will do it in [Chapter 3: Arrays and enumerations, Section 2: Enumerations](#).
- Combining all this knowledge into consistent code – you will do it in the final section of [Chapter 3: Array and enumerations, Section 4: Implementing initializers](#).

As you can see, at the beginning there is a lot to learn. Please don't run away and let me explain all those topics in a simple words. Please, give me a chance!

let and var keyword

To be able to make some experiments, please comment `Board` class body:

```
class Board {
    //private let rows, cols: Int
    //private var board: [[CellType]]
}
```

Declaring "variables" in Swift you have two options:

- With keyword `var` you can declare real *variable* – a components of your code whose contents you can change through a time. It may be

initiated as integer storing value 7 and later it can be changed to 3 or any other value you want.

- With keyword `let` you can declare *constant* (variable) – a components of your code whose contents, when initiated, may not be change through a time. If initiated as integer storing value 7 it cannot be changed and it keeps this value forever.

Remarks:

- Both variables and constants are called *field* or *property* if they are used in a class (for example, `rows` is a property of `Board` class).
- Despite formally we differ *variables* and *constants*, if this not lead to confusion, I will refer to both as *variables*.

Every object in Swift **must have** a *type* and this type must be known during compilation. Types in Swift fall into one of two categories:

- *value types*, where each **instance keeps a unique copy of its data**, usually defined as a `struct`, `enum`, or *tuple*.
- *reference types*, where instances share a single copy of the data, and the type is usually defined as a *class*.

Type defines kind of object allowed to be stored inside an object.

- You can *explicitly* define a type using *type annotation*:

```
let rows, cols: Int
var board: [[CellType]]
```

From this code Swift knows that both `rows` and `cols` are of integer type (`Int`) while `board` is a two-dimensional array (because of double square brackets pairs `[[]]`) of objects of type `CellType`.

- Type can be *inferred* based on the object substitution instruction. From the code below:

```
var foo = 1
```

Swift infer that `foo` variable must be an integer.

So, to avoid Type annotation missing in pattern error you have to provide either a type annotation when you declare constant or variable, or you have to assign a value to allow compiler infer the type.

In Swift every object **must have** a *value*. The following code possible to compile in C will not compile in Swift (I skip here the matter of different syntax):

```
#include <stdio.h>

int main() {
    int x;
    printf("%d", x);

    return 0;
}
```

In this code, first I declare variable `x` to be an integer and then I print it with `printf()`. Every time you run this code, you may see different result, a "random" result. Of course it is not random but `x` takes value which is stored in memory cell allocated for it. If you try to write equivalent code in Swift:

```
var x
print(x)
```

immediately you will see a Type annotation missing in pattern error. In some sense there is an option to have uninitialized variable in Swift if you explicitly define it as *optional*. Optional is a way to indicate that some object may not have a correct value. And this is important distinction: with optional you not say that some object has no value but rather you say that value it has is not a correct value of a given

type. This is a little bit tricky concept so I decided to devote a separate section discussing it.

At this moment, if you uncomment for a while variables located in **Board** class you will see `Class 'Board' has no initializers` message, which is the way Swift tells you that some properties in your class have no values, which is not acceptable by Swift. Even if you add an initializer to some of them but don't initialize all properties you will see another message: `Return from initializer without initializing all stored properties.`

Optionals

As you know from previous section, every object must have a value or must be explicitly defined as *optional* which is a way to indicate that this object may not have a value. You set an optional variable to a valueless state by assigning it the special value `nil`. In Swift, `nil` is used in case of the absence of a (correct) value of a certain type. You may say, that `nil` is a value which tells that there is no correct value. If it sounds a little bit crazy, recall `NaN` "number" which is a sequence of bits interpreted not as a number but as incorrect (non-existing) numeric value. Optionals of any type you want can be set to `nil`, not just object types. Conversely, `nil` cannot be used with non-optionals. If a variable (rather constant) in your code needs to take no value, you have to declare it as an optional value of the appropriate type.

If a variable (rather constant) in needs to take no value, you have to declare it as an optional value of the appropriate type.

Optional is denoted by question mark `?` suffixes type name: `Int?`, `Ship?` etc. The key question is: *How we can use optionals?* And the answer is not so obvious. We have few options.

DO NOTHING

Do nothing and treat optional as any other type:

```
var thisMayBeEmpty: Int?
```

```
print(thisMaybeEmpty)
thisMaybeEmpty = 1
print(thisMaybeEmpty)
thisMaybeEmpty = nil
print(thisMaybeEmpty)
```

In result you will see:

```
nil
Optional(1)
nil
```

Probably this is not what you want, as you have `Optional(1)` instead of `1`. Moreover, a warning message is displayed: Expression implicitly coerced from `Int?` to `Any` which is a clear signal that something is not used correctly.

FORCE UNWRAP

If you are sure (but you have to be sure if you don't want to crash your application) that optional contains a non-`nil` value, you can *force unwrap* its value with an exclamation mark `!` added at the end of the optional variable's name:

```
print(thisMaybeEmpty!)

if (thisMaybeEmpty != nil){
    print("Have some nonempty value: " + String(thisMaybeEmpty!))
} else {
    print("Nothing to print")
}
```

OPTIONAL BINDING WITH if-else

When working with optionals, `if-else` is something you have to use. There is a special syntax you can use in this context. *Optional binding* is used to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable:

```
if let nonempty = thisMaybeEmpty {
    print("Have some nonempty value: " + String(nonempty))
} else {
    print("Nothing to print")
}
```

```
}
```

Notice that for `nonempty` you don't have to use an exclamation mark.

If you chain together multiple optional bindings, the entire chain fails gracefully if any link in the chain is `nil`.

You can do even more: you can *chain together multiple optional bindings*, and the entire chain fails gracefully if any link in the chain is `nil`:

```
let number1: Int?
let number2: Int?

number1 = 2
number2 = nil

if let value1 = number1, let value2 = number2 {
  print(value1, value2)
} else {
  print("One of values is nil")
}
```

In this case you will see:

One of values is nil

USE `nil`-COALESCING OPERATOR

Another handy tool you can use is the *nil-coalescing operator* `??`. When used, for example in expression `(a ?? b)`, it unwraps an optional `a` if it contains a value, or return a default value `b` if `a` is `nil`. The expression `b` must match the type that is stored inside `a`. Executing the following code:

```
var thisMaybeEmpty: Int?
print(thisMaybeEmpty ?? 3)
```

you will see:

3

The nil-coalescing operator is a shorthand for:

```
a != nil ? a! : b
```

USE EXCLAMATION MARK !

1. How it was not so long time ago

Not so long time ago, when you were sure that optional had some value **every time you were going to use it**, you could get rid of the need to check and unwrap the optional's value every time it was accessed. It was possible thanks to assumption that you had a value all of the time when you needed it. These kinds of optionals were defined as *implicitly unwrapped optionals* (IUO in short) and you denoted them by placing an exclamation mark **!** instead of question mark **?** after the type you wanted to make optional. For example, this code

```
var implicitOptional: Int!  
print(implicitOptional)
```

in contrast to:

```
var thisMayBeEmpty: Int?  
print(thisMayBeEmpty)
```

didn't generate any warning or error and you could use **implicitOptional** variable without need of any explicit unwrapping with an exclamation mark:

```
implicitOptional = 1  
print(implicitOptional)
```

Implicitly unwrapped optional are fine if you know what you are doing. They require you to be absolutely sure there is a non-**nil** value before you use it. If you try to use a value that contains **nil**, your application will crash. You can't catch the error and you can't stop it from happening: your code will crash immediately.

2. How it is now

Today things has changed. When you use an exclamation mark **!** your code will compile but you will get a warning: `Coercion of implicitly unwrappable value of type 'Int?' to 'Any' does not unwrap optional along with number of possible fixes:`

- Provide a default value to avoid this warning
- Force-unwrap the value to avoid this warning
- Explicitly cast to 'Any' with 'as Any' to silence this warning

The warning comes from the way modern Swift handles IUO's. See [IOU:1-2] for the details. According to those documents, the appearance of **!** at the end of a property or variable declaration's type **no longer indicates that the declaration has IUO type**; rather, it indicates that

1. the declaration has optional type, and
2. the declaration has an attribute indicating that its value may be implicitly forced.

In consequence now you should consider **!** to be a **synonym** for **?** with the addition that it adds a flag on the declaration letting the compiler know that the declared value can be implicitly unwrapped if needed.

You can read `Int!` as: *this value has the type `Optional<Int>` as it would have in case of `Int?` declaration but also carries additional information saying that it can be implicitly unwrapped if needed*".

Because **!** tells the compiler that it can be implicitly unwrapped, so it can help ease in the need for optional chaining – see examples below.

Example 2.2.1

```
class A {
```

```

    var n: Int? = 1
}

class B {
    var a: A? = A()
}

let b: B? = B()
print(b?.a?.n ?? 0)

```

When you run this code, you will see:

1

Please note:

- I use default value 0 given after ?? operator in print statement to avoid Expression implicitly coerced from 'Int?' to 'Any' warning.
- The following will not compile as compiler will not implicitly unwrap optionals:

```
print(b.a.n)
```

Instead you will see the error: Value of optional type 'B?' must be unwrapped to refer to member 'a' of wrapped base type 'B'

Example 2.2.2

```

class A {
    var n: Int! = 1
}

class B {
    var a: A! = A()
}

let b: B! = B()
print(b.a.n)

```

Again, when you run this code, you will see:

1

Please note:

- In statement:

```
print(b.a.n)
```

compiler implicitly unwrap **!** as if I would write:

```
print(b!.a!.n)
```

- To avoid Coercion of implicitly unwrappable value of type 'Int?' to 'Any' does not unwrap optional warning you may type:

```
print(b.a.n!)
```

or

```
print(b.a.n ?? 0)
```

- You can still safely unwrap, if you want:

```
print(b?.a?.n)
```

As a final word, I can say it seems that coexistence of both **!** and **?** is temporal and one day **!** will be removed from Swift.

Functions

Swift functions can be characterized as follow (this list comprises only information you need in this book – full function characteristic is much more complex):

1. Each function parameter has both an *argument label* and *parameter name*.
2. You write an *argument label* **before** the *parameter name*, separated by a space.
3. The *argument label* is used when **calling** the function.
4. The *parameter name* is used in the **implementation** of the function.
5. By default, parameters use their parameter name as their argument label; in such case it is enough to specify only parameter name.
6. All parameters must have use unique names.
7. It is possible for multiple parameters to have the same argument label.
8. If you don't want to use an argument label for a parameter, an underscore character `_` must be used as label for that parameter.

9. If we use an underscore character, you cannot use a parameter name as an argument label.
10. If a parameter has an argument label, it must be used when you call the function.
11. Although arguments have their labels, you cannot change arguments order.

The following examples will clarify rules given above.

FUNCTION TAKING NO PARAMETERS

The simplest form of a function, taking no parameters and returning nothing, is as follow:

```
func functionSimplestForm(){  
    print("functionSimplestForm")  
}
```

and we call it as:

```
functionSimplestForm()
```

If you want to test it, you can simply paste this code into `main.swift` file:

```
import Foundation  
  
func functionSimplestForm(){  
    print("functionSimplestForm")  
}  
  
functionSimplestForm()
```

and run it; you will see in the **Console**:

```
functionSimplestForm
```

Program ended with exit code: 0

We may also explicitly define, with `Void` keyword following right arrow `->`, that function returns no value:

```
func functionSimplestFormVersion2() -> Void {  
    print("functionSimplestFormVersion2")  
}
```

but this is not Swifty style. Remember: *in Swift type only what is really necessary*. If your function *returns* something, for example `String`, this must be specified:

```
func functionReturningString() -> String {  
    return "Just a string"  
}  
  
print(functionReturningString())
```

If you forget to type `-> String`, Swift will complain: Unexpected non-void return value in void function.

FUNCTION WITH AN IMPLICIT RETURN

If the entire body of the function is a single expression, the function implicitly returns that expression so you don't have to type `return`:

```
func funcWithImplicitReturn(param1: Int, param2: Int) -> Int {  
    param1 * 3 + param2 * 5  
}
```

Of course if you want, you can type it but remember: *in Swift type only what is really necessary*:

```
func funcWithReturn(param1: Int, param2: Int) -> Int {  
    return param1 * 3 + param2 * 5  
}
```

FUNCTION WITH PARAMETERS

In the simplest form you specify only one *parameter name* (`param1`):

```
func functionWithOneParameter(param1: String) -> String {  
    return "Parameter value: " + param1  
}
```

and then call it as:

```
let result = functionWithOneParameter(param1: "foo")
```

If you missing argument label `param1` in call, you will get an error:
Missing argument label 'parameter1:' in call.

In case of more than one parameter, you write them separated by comma `,`:

```
func functWithMultipleParams(  
    param1: String,  
    param2: String  
) -> String {  
    return "Parameter value: " + param1 + " : " + param2  
}
```

and call as:

```
let result = functWithMultipleParams(  
    param1: "foo",  
    param2: "bar")
```

Notice how I format source code lines which are too long. Of course it is not a rule but it is a good habit to *keep your code neat, clean and consistent*.

RETURNING MULTIPLE VALUES

As in many programming languages, in Swift you can't return multiple values. It doesn't mean that you can't return multiple values. What?! Formally, every function may return only one thing. This *thing* could be a primitive type and then you will return only one value. *Thing* could be also a complex type, for example class, struct, tuple or array. In such a case you can pack multiple values into complex type. Then formally you return only one *thing* with multiple values "hidden" inside.

The following example shows how you can use *tuples* to return two values (I explain you tuples in [Chapter 5: Tuples, switch and extensions, Section 1: Tuples](#)):

```
func functionWithMultipleReturnValues(
    val1: Int,
    val2: Int
) -> (sum: Int, product: Int) {
    let sum = val1 + val2
    let prod = val1 * val2

    return (sum, prod)
}

let result = functionWithMultipleReturnValues(val1: 2, val2: 3)
let s = result.sum
let p = result.product
```

HOW TO USE ARGUMENT LABEL AND PARAMETER NAME

Example below shows all possible combinations of argument label and parameter name usage (in comment given after two slashes `//` I put number of "rule" applying to given line):

```
func functionLabelsTest(
    argumentLabel parameterName: String, // 2
    parameterWithDefaultLabel: String, // 5
    _ parameterWithNoArgumentLabel: String, // 8
    justALabel parameter1: String, // 6, 7
    justALabel parameter2: String // 6, 7
) {
    print(
        parameterName + ":" + // 4
```

```

    parameterWithDefaultLabel + ":" + // 4
    parameterWithNoArgumentLabel + ":" + // 4
    parameter1 + ":" + parameter2) // 4
}

functionLabelsTest(
    argumentLabel: "first", // 3, 10
    parameterWithDefaultLabel: "second", // 5
    "third", // 9
    justALabel: "fourth", // 3, 10
    justALabel: "fifth") // 3, 10

```

You may ask, why do we have both argument label and parameter name? My answer is: to make developers life easier and to *keep your code neat, clean and consistent*. Argument label is for callee – it should be descriptive so the person who calls function knows for what argument is used, what it should be etc. Notice that correct function name in Swift contains both function name and all argument labels. So

`functionLabelsTest`

is not correct; you should say:

```

functionLabelsTest(argumentLabel:parameterWithDefaultLabel:_:ju
stALabel:justALabel:)

```

Every argument label should correspond to function name and other labels so when you read all of them as a one sentence it should sound as a one sentence. Because argument labels are only hints for callee, and all arguments must preserve their order (see rule 11), there is no danger of confusing them with each other.

On the other hand, parameter name is used internally, in the function body. The one who uses it knows what they are doing and why so this name don't have to be descriptive but must allow the parameters to be clearly distinguished from each other so they may be used in implementation.

Without any example it may still sound strange, so below I'm giving one which will clarify the case:

```
func greeting(for who: String) -> String {  
    return "Hello, " + who + "!"  
}  
print(greeting(for: "Piotr"))
```

In this case correct function name is neither

`greeting`

nor

`greeting(who:)`

but

`greeting(for:)`

For me, the last one sounds best and this is why in Swift you may use two different names for the same thing: first is used outside and should "sound good" while second is visible only internally.

NOTE

Argument or parameter?

The terms *parameter* and *argument* are sometimes used interchangeably, and the context is used to distinguish the meaning. The term **parameter** (sometimes called **formal parameter**) is often used to refer to the variable as found in the function definition, while **argument** (sometimes called **actual parameter**) refers to the actual

input passed. For example, in the function definition $f(x) = 2x$ the variable x is a parameter; in the function call $f(2)$ the value 2 is the argument of the function. Loosely, a parameter is a *type*, and an argument is an *instance*.

Please keep in mind that for example in the classical *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie book, in section 5.10 *Command-line Arguments* you can find the following code:

```
#include <stdio.h>
main(int argc, char *argv[])
{
    ...
}
```

As you can see there is no `paramc` but `argc` and not `paramv` but `argv`.

Naming the first parameter `argc` isn't a mistake or error. At run time the value you use is an *argument*. We reserve the term *parameter* for situations when discussing subroutine definitions.

According to authors explanation of this part of code: *When main is called, it is called with two arguments. The first (conventionally called argc, for argument count) is the number of command-line arguments the program was invoked with; the second (argv, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.*

Swift elegantly solves this naming problem using both argument label and parameter name.

PARAMETERS WITH DEFAULT VALUES

There are situations when function *in most cases* is called with the same parameters values. If it is also in your case, you can make other developer's life easier and provide default values for those parameters. If they want, they may override them, but if default values work for them it's fine.

```
func fooFunction(  
    noDefaultParameter: String,  
    defaultParameter: String = "defaultValue"  
) {  
    return noDefaultParameter + " " + defaultParameter  
}  
  
fooFunction(noDefaultParameter: "mustBeDefinedByUser")  
fooFunction(  
    noDefaultParameter: "mustBeDefinedByUser",  
    defaultParameter: "alsoDefinedByUser")
```

If you have default parameters, you can omit any default parameter you want:

```
func funcWithDefaults(  
    param1: Int = 2,  
    param2: Int = 3  
) {  
    print(param1 * 3 + param2 * 5)  
}  
  
funcWithDefaults()  
// Prints: 2 * 3 + 3 * 5 = 6 + 15 = 21  
  
funcWithDefaults(param1: 4)  
// Prints: 4 * 3 + 3 * 5 = 12 + 15 = 27  
  
funcWithDefaults(param2: 5)  
// Prints: 2 * 3 + 5 * 5 = 6 + 25 = 31
```

VARIADIC PARAMETERS

A *variadic parameter* accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be

passed a varying number of input values when the function is called. You declare variadic parameters by inserting three period characters `...` after the parameter's type name.

The values passed to a variadic parameter are made available within the function's body as an array of the appropriate type. For example, a variadic parameter with a name of `numbers` and a type of `Int...` is made available within the function's body as a constant array called `numbers` of type `[Int]`. I will explain you arrays in [Chapter 3: Arrays and enumeration, Section 1: Arrays](#), but I hope the following example will. be intuitively understandable for you.

Note that the first parameter that comes after a variadic parameter must have an argument label to avoid any ambiguity which arguments are passed to the variadic parameter and which arguments are passed to the parameters that come after the variadic parameter.

Starting of Swift 5.4 a function can have multiple variadic parameters.

Now you can see how it works in real example:

```
func calculate(
    sumOf numbersSequence1: Int...,
    productOf numbersSequence2: Int...,
    multipliedBy multiplier: Int
) -> (sum: Int, product: Int) {
    var sum = 0
    for number in numbersSequence1 {
        sum += number
    }

    var product = 1
    for number in numbersSequence2 {
        product *= number
    }

    return (sum * multiplier, product * multiplier)
}

let result = calculate(sumOf: 6, 5, 4, productOf: 3, 2, 1,
    multipliedBy: 2)
print(result.sum)
```

```
// Prints:  
// 30  
print(result.product)  
// Prints:  
// 12
```

IN-OUT PARAMETERS

Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. If you want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, define that parameter as an *in-out parameter* instead.

You write an in-out parameter by placing the `inout` keyword right before a parameter's type. You place an ampersand `&` directly before a variable's name when you pass it as an argument to an in-out parameter, to indicate that it can be modified by the function.

Here's a typical example of a swap function which interchanges two numbers:

```
func swap(integer a: inout Int, withInteger b: inout Int) {  
    let temp = a  
    a = b  
    b = temp  
}  
  
var x = 3, y = 4  
swap(integer: &x, withInteger: &y)  
print(x)  
// Prints: 4  
  
print(y)  
// Prints: 3
```

UNWANTED RESULTS

Sometimes, this is rare but may happen, you may not need result returned by a function:

```
func doSomethingWithInteger(arg: Int) -> Int {
    let newValue = 2 * arg

    print(newValue)

    return newValue
}

doSomethingWithInteger(arg: 3)
```

If you leave code as it is given above, you will have to tolerate annoying warning: Result of call to 'doSomethingWithInteger(arg:)' is unused. This warning is a valuable information for developer: *This function returns something but you don't use it. Maybe this is OK, but maybe you forget about something.* If you really don't want to use result, you have to tell it to Swift with underscore mark `_`:

```
_ = doSomethingWithInteger(arg: 3)
```

You can treat it as a way to say: *I don't care about this value.*

THIS IS NOT THE END

This is not the whole story about functions. Some topics are still untouched:

1. How to use function as any other type.
2. What nested functions are.
3. What closures are, how we can use them.

I will discuss them in the next section. If it is first time you read about Swift you may consider those topics as too complicated. You may safely skip next chapter now and return when you will be ready to know more about functions.

More about functions – closures

At first read you may safely skip this chapter and return when you will be ready to know more about functions.

Functions as type

Every function in Swift has a specific *function type* consisting of the parameter types and the return type of the function.

For example, function defined as:

```
func foo(){  
    // Some code goes here  
}
```

is of the type `() -> Void` while function:

```
func bar(_ a1:String, _ a2: String) -> String {  
    // Some code goes here  
}
```

is of the type `(String, String) -> String`. A function type is used just like any other type:

```
var action: (Int, Int) -> Int // Function type  
func add (x: Int, y: Int) -> Int {
```

```

    return x + y
}

func sub (x: Int, y: Int) -> Int {
    return x - y
}

action = add

print(add(x: 3, y: 5))
// Prints: 8
print(action(3, 5))
// Prints: 8

```

Notice that despite both `add(x:y:)` and `sub(x:y:)` have parameters `x` and `y`, and you have to use parameters name when you call directly `add(x:y:)` or `sub(x:y:)`, you call `action` without any parameters name; otherwise you will get a Extraneous argument labels `'x:y:'` in call compilation error.

You can use function type as a parameter type for another function:

```

func doSomethingWithTwoInts(
    _ int1: Int,
    _ int2: Int,
    _ action: (Int, Int) -> Int
) {
    let result = action(int1, int2)
    print(result)
}

doSomethingWithTwoInts(3, 5, add)
// Prints: 8

```

An interesting question is: *How you can use a function type as the return type of another function?* An answer: *As any other existing type.* For example, the function `selectAction(_)` defined as:

```

func selectAction(
    _ decision: String = "add"
) -> ((Int, Int) -> Int) {
    switch decision {
    case "sub":
        return sub
    default:

```



```

    return add
}
}

```

returns a function of type `(Int, Int) -> Int`. You can use this function as it is showed below:

```

action = selectAction()
print(action(3, 5))
// Prints: 8

// or

print(selectAction()(3, 5))
// Prints: 8

```

Nested functions

Swift allowed you to do more with functions: it can be seen as strange and awkward but you can define functions inside the bodies of other functions – this way you have so called *nested functions* to distinguish from previously discussed *global functions*. Nested functions are hidden from the outside, but of course can be called by and used by enclosing function. An enclosing function can also return one of its nested functions which allow the nested function to be used in another scope.

As an example you can reimplement `selectAction(_)` function "embedding" `add(x:y)` and `sub(x:y:)` inside:

```

func selectAction(
    _ decision: String = "add"
) -> ((Int, Int) -> Int) {

    func add(x: Int, y :Int) -> Int {x + y}
    func sub(x: Int, y: Int) -> Int {x - y}

    switch decision {
    case "sub":
        return sub
    default:
        return add
    }
}

```

```
}  
  
action = selectAction("sub")  
print(action(4,5))  
// Prints: -1
```

Closures

Closures are self-contained blocks of code that can be passed around and used in your code. Closures in Swift are similar to blocks in Objective-C and to lambdas in other programming languages. In short, closures are Swift's anonymous functions.

Closures are a technique for implementing lexically scoped name binding. You can think of it as a "box" storing a function together with an "environment" needed to execute this function. Each variable that is used locally, but defined in an enclosing scope is associated with the value or storage location to which the name was bound when the closure was created. What is important, a closure, unlike a plain function, allows the function to access those captured variables through the closure's reference to them, even when the function is invoked outside their scope. Don't worry if this explanation doesn't explain you anything. The subsequent subsection clarifies general idea behind closures.

GENERAL IDEA

To understand how it works, let's analyze the following example:

```
var f: () -> Int  
func constant() -> Int { 0 }  
f = constant  
  
print("Call to f(): \(f())") // 1  
  
func foo() {  
    var x=1 // 2  
    func nested() -> Int { x += 1; return x } // 3
```

```

    f = nested // 4
    let result = f() // 5
    print("Inside foo, call to f(): \(result)") // 6
}

foo() // 7
print("Call to f(): \(f())") // 8

```

In line marked with comment `// 3`:

```
func nested() -> Int { x += 1; return x }
```

you define a nested function. Variable `x` is defined outside of this function in line `// 2` but is used inside. So, you can say, that this function needs `x` to work correctly. And there is nothing surprising that call:

```
foo() // 7
```

and in consequence:

```
let result = f() // 5
```

works and returns correct value (line marked with `// 6`):

Inside foo, call to f(): 2

The question is how it is possible that code at line `//8`:

```
print("Call to f(): \(f())")
```

also works?! Now you are outside of `foo()` function and you try to call `f()`. How it could be that `f()` knows value of the `x` variable which is visible only within the scope of `foo()` function? This is when *closure* comes into play.

The assignment statement:

```
f = nested // 4
```

"saves" function along with all the outer environment needed to execute this function (in this case, `x` variable).

Consider a second example:

```
func referenceValue(_ x: Int) -> ((Int) -> (Int)){
    func innerFunction(y: Int) -> Int {
        return y + x
    }
    return innerFunction
}

var moveFrom3By = referenceValue(3)
var moveFrom5By = referenceValue(5)

print(moveFrom3By(7))
// Prints: 10
print(moveFrom5By(9))
// Prints: 14
```

The above code defines a function `referenceValue(_)` with a parameter `x` and a nested function `innerFunction(y)`. The nested function has access to `x`, because it is in the *lexical scope* of `x` (note that `x` is not local to this function). The function `referenceValue(_)` returns a closure containing:

1. the inner `innerFunction(y)` function, which adds the `y` value to the `x` (reference) value;
2. and a reference to the variable `x` from this invocation of `referenceValue(_)`, so inner function will know where to find it when invoked.

Note that `referenceValue(_)` returns a function. This means that both `moveFrom3By` and `moveFrom5By` are of function type, so you can invoke `moveFrom3By(7)` and `moveFrom5By(9)`. Interesting is that

while `moveFrom3By` and `moveFrom5By` refer to the same anonymous function, the associated environments differ, and invoking the closures will bind the name `x` to two distinct variables with different values (`3` and `5`) in the two invocations, thus evaluating the function to different results (`10` and `14`).

BASIC USAGE

General closure expression syntax has the following form:

```
{(parameters) -> returnType in
  expressions
}
```

This syntax is very close to function syntax but without name (for this reason we say about *anonymous functions*). The parameters can be in-out, named variadic parameter and tuples. We cannot use default values for them.

Having previously defined `doSomethingWithTwoInts(_:_:)` function you can use it as follow:

```
doSomethingWithTwoInts(
  5,
  7,
  {(a: Int, b: Int) -> Int in return a + b}
)
// Prints: 12
```

In this example the third parameter is an inline closure expression:

```
{(a: Int, b: Int) -> Int in
  return a + b
}
```

Because it is always possible to infer the parameter and return type when passing a closure to a function or method as an inline closure expression, so you never need to write an inline closure in its full form when the closure is used as a function or method argument. In consequence you can write the above in a short form as:

```
doSomethingWithTwoInts(5, 7, {a, b in return a + b})
```

What is more, *single expression closure* can implicitly return the result by omitting the `return` keyword from declaration (as you do for "normal" functions):

```
doSomethingWithTwoInts(5, 7, {a, b in a + b})
```

This is not the end, because even this short expression can be shorthand! In Swift a shorthand argument names `$0`, `$1` and so on can be used to refer to the values of the closure's first, second and so on arguments. If the argument is a tuple `$0.0` is a key, and `$0.1` is a value of the closure's first argument.

```
doSomethingWithTwoInts(5, 7, {$0 + $1})
```

If you think that there is no option to write shorter expression, you are wrong. You can use *operator methods*. From their definition Swift can infer the number and the types of arguments and return value:

```
doSomethingWithTwoInts(5, 7, +)
```

If the closure is a final argument of a function you are going to use, and the closure expression is long, you can write it as a *trailing closure*. A *trailing closure* is a closure which is written after the function call's parentheses (but it is still an argument to the function):

```
doSomethingWithTwoInts(5, 7){a, b in  
    a + b  
}
```

or

```
doSomethingWithTwoInts(5, 7){$0 + $1}
```

If the closure expression is provided as the function's or method's only argument and you provide that expression as a trailing closure, then you

don't need to write a parentheses after the function's or method's name when you call it.

In the example below you will see very basic but useful example of closure usage. Please have in mind that in the code below you will create two constants, which in some sense will behaves like variables. This is because functions and closures are reference types. So event if we define a constant related with a closure this relation is constant (doesn't change) but the closure this constant refers to is still able to change some captured variables.

```
func getIncrementer(incrementBy value: Int) -> (() -> Int) {
    var incrementerValue = 0;

    func incrementer() -> Int {
        incrementerValue += value
        return incrementerValue
    }

    return incrementer
}

let incrementerBy_plus_2 = getIncrementer(incrementBy: 2)
let incrementerBy_minus_2 = getIncrementer(incrementBy: -2)

print(incrementerBy_plus_2())
// Prints: 2
print(incrementerBy_minus_2())
// Prints: -2
print(incrementerBy_plus_2())
// Prints: 4
print(incrementerBy_minus_2())
// Prints: -4
print(incrementerBy_plus_2())
// Prints: 6
print(incrementerBy_minus_2())
// Prints: -6
```

MULTIPLE CLOSURES

It is perfectly legal for a function to take more than one closure:

```
func doSomethingWithTwoInts(
    _ int1: Int,
    _ int2: Int,
```

```

    _ action: (Int, Int) -> Int?,
    onSuccess: ((Int) -> Void)?,
    onFailure: ((Int, Int) -> Void)?
) -> Int? {
    if let result = action(int1, int2) {
        if let onSuccess = onSuccess {
            onSuccess(result)
        }
        return result
    } else {
        if let onFailure = onFailure {
            onFailure(int1, int2)
        }
        return nil
    }
}

```

In this case you don't want to only execute some action on two integers but also allow user to make an action in case of success or (what is more important) any problems. When your function takes multiple closures, you omit the argument label for the first trailing closure and you label the remaining trailing closures. For example, if you implement `divide` function:

```

func divide (x: Int, y: Int) -> Int? {
    if y != 0 {
        return x/y
    }
    return nil
}

```

you may try to use it as an action in

`doSomethingWithTwoInts(_:_:_:onSuccess:onFailure:)` function along with some additional actions specific to division:

```

var statusDescription = ""

var result = doSomethingWithTwoInts(15, 0, divide){_ in
    statusDescription = "Action completed"
}
onFailure: {int1, int2 in
    statusDescription =
        ""
        It's impossible to divide \(int1) by \(int2)
        You can try to divide \(int2) by \(int1)
}

```



```

    ....
}

if let result = result {
    // Result is correct
    // Do something with it
    print(result)
} else {
    print(statusDescription)
}

```

In this case additional action is justified as division is not always possible to be performed:

It's impossible to divide 15 by 0
You can try to divide 0 by 15

On the other hand, addition of two integers is always possible, so there is no need to perform any additional actions in case of failure:

```

result = doSomethingWithTwoInts(
    15,
    0,
    add,
    onSuccess: nil,
    onFailure: nil
)

print(result!)
// Prints:
// 15

```

ESCAPING CLOSURES

A closure is said to *escape a function* when the closure is passed as an argument to the function, but is called *after* the function returns. By prefix any closure argument with **@escaping**, you convey the message to the caller of a function that this closure *can outlive* (escape) the function call scope. By default a closure is non-escaping, and its lifecycle end along with function scope.

The following is an example of a non-escaping closure:

```
func processWaitForResult(
    data: ProcessingData,
    action: (ProcessingData) -> ResultData
) -> ResultData {
    let result = action(data)
    return result
}
```

The closure passed to this function is executed immediately before a result is returned. Other words, function waits for a result to be able to return it and ends its (function) execution. Because the closure is executed within function scope and before function ends, you know that nothing you do inside of the closure can leak or outlive the scope of the function.

If you define **ProcessingData**, **ResultData** and some functions possible to call:

```
class ProcessingData {
    // Some code goes here
    var data = 0
}

class ResultData {
    // Some code goes here
    var data = 0
}

func process1(_ data: ProcessingData) -> ResultData {
    let result = ResultData()
    result.data = data.data + 1
    return result
}

func process2(_ data: ProcessingData) -> ResultData {
    let result = ResultData()
    result.data = data.data * 2
    return result
}

func process3(_ data: ProcessingData) -> ResultData {
    let result = ResultData()
    result.data = data.data * data.data
    return result
}
```

you may execute `processWaitForResult(data:action:)` as:

```
let data = ProcessingData()
data.data = 3
let result = processWaitForResult(data: data, action: process3)
print(result.data)
// Prints:
// 9
```

The question is: *When closure escapes?* One of the simplest way that a closure can escape is by being stored from a function in a variable that's defined outside the function (`batchQueue`):

```
var batchQueue = [(ProcessingData) -> ResultData]()

func addProcessingStep(
    action: @escaping (ProcessingData) -> ResultData
) {
    batchQueue.append(action)
}
```

In this case you have to add `@escaping` keyword to avoid a compile-time error: Converting non-escaping parameter 'closure' to generic parameter 'Element' may allow it to escape. and to clearly indicate your intention. You use the variable `batchQueue` to allow closures survive when function ends.

Now you may define a method executing function stored in `batchQueue` in provided order:

```
func batchProcessing(data: ProcessingData) -> ResultData? {
    // Guard (safety check)
    if batchQueue.count == 0 {
        return nil
    }

    var result = batchQueue[0](data)

    for i in 1..
```

```
// Result of the processing in one step may be used as
// the input data for the next processing step
func convertResultToData(
    _ result: ResultData
) -> ProcessingData {
    // Do some conversion here
    let data = ProcessingData()
    data.data = result.data
    return data
}
```

You may execute `batchProcessing(data:)` as:

```
addProcessingStep(action: process1)
addProcessingStep(action: process2)
addProcessingStep(action: process3)

result = batchProcessing(data: data)!
print(result.data)
// Prints:
// 64
```

NOTE

Note 1: It's very common to use escaped closures as a *completion handler* during asynchronous operations. In this case the function returns after it starts the operation, but the closure isn't called until the operation is completed. The closure needs to escape, to be called later.

Note 2: An escaping closure that refers to `self` needs special consideration every time `self` refers to an instance of a class. Capturing `self` in an escaping closure makes it easy to accidentally create a *strong reference cycle*. This is related to Automatic Reference Counting which is out of the scope of this book.

Code changes summary

At this moment you have a "stub" or skeleton of a **Board** class with two constants and one variable inside. Apart of that, if both variable and constants are uncommented, you also have two errors:

- `Class 'Board' has no initializers`
- `Cannot find type 'CellType' in scope`

Happily, having the knowledge from the current chapter, extended with information from the next one, you will easily fix this.

Arrays and enumerations

You will do:

In this part you will create a very basic classes with properties, initializers and methods. One of them you will use to represent game board and another one to be an entry point to the whole game logic.

You will learn:

- How to create and use **arrays**.
- How to define **enumeration**.
- What **range operators** are.

Arrays

The *array* stores values of **the same type in an indexed order**. Every value is uniquely indicated by an *index* determining its *position*. The same value can appear in an array multiple times at different positions.

One-dimensional arrays

You *declare* or *define* one-dimensional array this way:

```
// Declare an array
var mutableArrayDeclaration: [String]
let immutableArrayDeclaration: [String]

// Define an empty array
var mutableArray1 = [String]()
var mutableArray2 = Array<String>()
let immutableArray = [String]()
```

With declaration you only say: *One day this would be an array*. With definition you say: *This is an array*. If your array is defined, it exists, you can add something to it:

```
// You can do
mutableArray1.append("Zero")
mutableArray1.append("One")
mutableArray1.append("Two")
// or
mutableArray2 = ["zero", "one", "two"]
// but can't do
//mutableArrayDeclaration.append("Zero")
```

```
// or
//immutableArray.append("Zero")

print(mutableArray1)
// Prints: ["Zero", "One", "Two"]
print(mutableArray2)
// Prints: ["zero", "one", "two"]
mutableArray1[1] = "NEW"
print(mutableArray1)
// Prints: ["Zero", "NEW", "Two"]
```

Great! – you may think. It seems to be dynamic structure as I haven't declared any array size but was able to add (append) elements. You will be disappointed if you try to get an access to any array's element you want. The following code is perfectly legal in PHP:

```
<?php
$array = array();
$array[5] = 3;

print_r($array);
/* Prints:
Array
(
    [5] => 3
)
*/
?>
```

In Swift, if you try this code:

```
mutableArray1[5] = "NEW"
print(mutableArray1)
```

you will get a run time error: Fatal error: Index out of range. If you remember what I said in the [Chapter 2: Variables and functions, Section 1: Variables](#) about variables, you shouldn't be surprised: in Swift every object **must have** a type and **value**. So initially `mutableArray1` exists (is not `nil`) but has no values – is declared as an array of size 0. When you append something, behind a scene, Swift creates **new** array with size extended to be able to append new elements. In this sense it is

dynamic. But all the time it has strictly defined number of elements so you cannot freely get access at any index you want; otherwise you will "jump" out of the range of possible indices.

ARRAY WITH PREDEFINED SIZE

In Swift, differently than in C, we can't create an array that has pre-allocated memory but does not contain elements. The following C code compiles and executes without any problems:

```
#include <stdio.h>

int main() {
    int x[10];
    x[5] = 4;
    printf("%d\\t%d", x[5], x[2]);
    // Prints: 4 0

    return 0;
}
```

You can't create an *empty* array of fixed, predefined size; instead you can create an array of predefined size filled with specified element – remember: in Swift every object **must have** a type and **value**:

```
var mutableArray3 = [String](repeating: "", count: 5)
print(mutableArray3)
// Prints: ["", "", "", "", ""]

mutableArray3[3] = "THREE"
print(mutableArray3)
// Prints: ["", "", "", "THREE", ""]
```

ARRAYS CONCATENATION AND RANGES USAGE

A good news is that you can easily concatenate arrays:

```
mutableArrayDeclaration = mutableArray2 + mutableArray3

print(mutableArrayDeclaration)
// Prints:
// ["zero", "one", "two", "", "", "", "THREE", ""]
```

```
mutableArrayDeclaration += ["FIVE", "SIX"]

print(mutableArrayDeclaration)
// Prints:
// ["zero", "one", "two", "", "", "", "THREE", "", "FIVE",
"SIX"]
```

You can do even more – with subscript syntax you can change a range of values at once, even if the replacement set of values has a different length than the range we are replacing. The number of indices defined by a range may be lower than number of elements you want to use:

```
mutableArrayDeclaration[1...2] = ["ONE", "TWO", "THREE"]
print(mutableArrayDeclaration)
// Prints:
// ["zero", "ONE", "TWO", "THREE", "", "", "", "THREE", "",
"FIVE", "SIX"]
```

In this case index range `1...2` define two indices: `1` and `2`, while you try to use array with three elements. In such a case Swift starts at first index given by your range and, if necessary, extends array to put all new values to a new array (previous array had 10 elements, while current has 11 – two elements was replaced by three, so there is one element more). Opposite, the number of indices defined by a range may be greater than number of elements you want to use:

```
mutableArrayDeclaration[4...6] = ["***"]
print(mutableArrayDeclaration)
// Prints:
// ["zero", "ONE", "TWO", "THREE", "***", "THREE", "", "FIVE",
"SIX"]
```

In this case index range `4...6` define three indices: `4`, `5` and `6`, while you try to use array with only one element. In such a case Swift starts at first index and replace all subsequent elements as long as there are values in replacement array. Next all elements without defined replacement are removed (previous array had 11 elements, while current has 9 – three elements was replaced by one element, so there are two elements less).

Removing element is straightforward:

```
mutableArrayDeclaration.remove(at: 4)
print(mutableArrayDeclaration)
// Prints:
// ["zero", "ONE", "TWO", "THREE", "THREE", "", "FIVE", "SIX"]
```

ITERATING OVER AN ARRAY

Random access to array is great but sometimes you want to iterate over every element in your array. For this purpose, you can use **for loop**:

```
for item in mutableArrayDeclaration {
    print("value: \(item)")
}
/*
value: zero
value: ONE
value: TWO
value: THREE
value: THREE
value:
value: FIVE
value: SIX
*/

for (index, value) in mutableArrayDeclaration.enumerated() {
    print("at index: \(index) value: \(value)")
}
/*
at index: 0 value: zero
at index: 1 value: ONE
at index: 2 value: TWO
at index: 3 value: THREE
at index: 4 value: THREE
at index: 5 value:
at index: 6 value: FIVE
at index: 7 value: SIX
*/
```

Two dimensional arrays

In Swift, as in many other programming languages, two-dimensional array of objects is a one-dimensional array of one-dimensional-arrays of objects. And n-dimensional array of objects is a one-dimensional array of one-dimensional-arrays of... etc. To creating a multi-dimensional array you have to add another set(s) of square brackets `[` and `]`. For example, to turn one-dimensional array of integers `[Int]` into a two-dimensional array (array of arrays), you should just write `[[Int]]`:

```
var array2DEx1 = [[Int]]()
```

Now you can create three simple 1D arrays of integers:

```
var row1 = [11, 12, 13, 14]
var row2 = [21, 22, 23]
var row3 = [31, 32]
```

Next, you can add them all to the "main" array:

```
array2DEx1.append(row1)
array2DEx1.append(row2)
array2DEx1.append(row3)
print(array2DEx1)
// Prints:
// [[11, 12, 13, 14], [21, 22, 23], [31, 32]]
```

This allows you to operate on existing elements:

```
array2DEx1[1].append(24)
print(array2DEx1)
// Prints:
// [[11, 12, 13, 14], [21, 22, 23, 24], [31, 32]]

array2DEx1[1][1] = 0
print(array2DEx1)
// Prints:
// [[11, 12, 13, 14], [21, 0, 23, 24], [31, 32]]
```

ARRAY WITH PREDEFINED SIZE

To create multi-dimensional array with predefined size, you simply embed required number of time array's initializer `init(repeating: Element, count: Int)`:

```
var array2DEx2: [[Int]] = []
array2DEx2 = Array(repeating: Array(repeating: 0,
                                     count: 2),
                  count: 3)

// or
//var array2DEx2: [[Int]] = Array(repeating: Array(repeating: 0,
//                                                  count: 10),
//                                count: 10)
// or
//var array2DEx2 = Array(repeating: Array(repeating: 0,
//                                         count: 10),
//                        count: 10)
print(array2DEx2)
// Prints:
// [[0, 0], [0, 0], [0, 0]]

array2DEx2[1][1] = 1
print(array2DEx2)
// Prints:
// [[0, 0], [0, 1], [0, 0]]

var array2DEx3 : [[Int]] = [[1, 2, 3], [4, 5, 6]]
print(array2DEx3)
// Prints:
// [[1, 2, 3], [4, 5, 6]]
```

Array is a value type

Array in Swift is a value type. It means that **each array instance keeps a unique copy of its data**. Swift creates a copy of an array whenever you define a variable based on some existing array:

```
var array = [1, 2, 3, 4, 5] // 1
var arrayCopy = array      // 2
array[0] = 0               // 3
print(array)               // 4
// Prints "[0, 2, 3, 4, 5]"
```

```
print(arrayCopy)           // 5
// Prints "[1, 2, 3, 4, 5]"
```

Below I explain all the steps:

- In the first line (line marked as 1) you define array of integers with five elements.
- In the second line you define new variable `arrayCopy` based on previously defined array. Now `arrayCopy` is a copy of `array`. In consequence, integer 1 at index 0 in `arrayCopy` is a different object than integer 1 at index 0 in `array` – as a copy, it occupies different memory area but in this area stores the same binary sequence (sequence coding integer of value 1).
- You prove it in line number three where you change value of `array` at index 0.
- Now when you print `array` (line number four) you see [0, 2, 3, 4, 5], while printing `arrayCopy` (line number five) you see "initial" values: [1, 2, 3, 4, 5].

In the second example I use a reference type to fill array with values:

```
class referenceType {
    var value = 1
}
var array = [referenceType(), referenceType()]
var arrayCopy = array
```

Also in this case, as for integers above, `arrayCopy` is a copy of `array`. Now again `referenceType` at index 0 in `arrayCopy` is a different object than `referenceType` at index 0 in `array` – as a copy, it occupies different memory area but in this area stores the same binary sequence (sequence coding pointer to `referenceType`). Both objects are different, but because both are reference type, in consequence both

point the same object. Modifications to an instance (an element of array) are visible from either array:

```
array[0].value = 0
print(array[0].value)
// Prints: 0
print(arrayCopy[0].value)
// Prints: 0
```

Modifications to an array are again visible only in the modified array:

```
array[0] = referenceType()
print(array[0].value)
// Prints "0"
print(arrayCopy[0].value)
// Prints "1"
```

Remember, Swift also makes a copy of an array every time you pass an array as an argument of a function. This is something totally different than you may know from for example C programming language. The following code compiles and executes without any problems:

```
#include <stdio.h>

void display(int array[]) {
    array[0] = 0;
}

int main() {
    int array[] = {1, 2, 3, 4, 5};

    printf("%d\t", array[0]);
    // Prints: 1

    display(array);

    printf("%d", array[0]);
    // Prints: 0
    return 0;
}
```

Notice that contents of an array has changed as a result of call to **display** function.

If you try to write similar code in Swift:

```
func display(_ array: [Int]) {  
    array[0] = 0;  
}  
  
var array = [1, 2, 3, 4, 5]  
print(array[0]);  
// Prints: ?  
display(array);  
print(array[0]);  
// Prints: ?
```

you will get an error `Cannot assign through subscript: 'array' is a 'let' constant and it will not even compile`. If `array` inside a function is a `let` constant it must be a copy of original `array` (defined outside of `display` function) which is a `var` variable.

In reality, for performance reasons, the Swift compiler tries to avoid unnecessary copying whenever possible – this is a common sense approach, as arrays may have a lot of elements and making a real copy is a resource (time and memory) consuming process. Even if it says that an array is officially copied, it doesn't mean that it is really copied. Arrays, like all other collections of variable size, use copy-on-write optimization. Multiple copies of an array share the same storage until you modify one of the copies. When that happens, the array being modified replaces its storage with a uniquely owned copy of itself, which is then modified in place.

This means that if an array is sharing storage with other copies, **the first mutating operation on that array incurs the cost of copying the array**. An array that is the sole owner of its storage can perform mutating operations in place.

Enumerations

In Swift enumerations are much like Java's enumerations. They incorporate many features usually supported by classes, such as (we will show all of them in class section of Swift tutorial)

Enumeration (or simply *enum*) is a user defined data type, mainly used to assign names to integer constants. The idea behind enums is to replace all "magic numbers" – meaningless numbers, with descriptive names. The names used in place of numbers make a program easy to read and maintain, enables you to work with those values in a type-safe way within your code.

In Swift enumerations are much like Java's than C enumerations. They incorporate many features usually supported by classes, such as:

- computed properties (see [Chapter 6: Properties, dictionaries and sets, Section 1: Property types, Subsection: Computed properties](#)) to provide some additional informations about the current value of enumeration;
- instance methods to provide functionality related to the values the enumeration represents;
- initializers;
- can be extended;

- can conform to protocols.

At the introductory level of this book, I will jump over all enumeration's advanced features and will show you the most basic usage – as a "mapper" from name to integer.

After declaring enumeration:

```
enum Planet {  
    case Mercury, Venus, Earth, Mars,  
        Jupiter, Saturn, Uranus, Neptune  
}
```

you can use it to declare or define variables or constants:

```
var currentPlanet: Planet  
var selectedPlanet = Planet.Earth
```

The type of `selectedPlanet`, as for all other types, is inferred while it is initialized. Once a variable is declared as type of `Planet`, you can set it to a different `Planet` value:

```
selectedPlanet = .Mars  
  
switch selectedPlanet {  
case .Earth:  
    print("You can live here")  
case .Mars:  
    print("Maybe one day you can live here")  
default:  
    print("No chance to live here")  
}  
// Prints:  
// Maybe one day you can live here
```

By default first case item from enum corresponds to 0, second item corresponds to 1 and so on. If necessary, you can change default values of enum elements during declaration:

```
enum Planet2: Int {  
    case Mercury = -2, Venus, Earth, Mars,
```

```

        Jupiter, Saturn = 12, Uranus, Neptune
    }

    let earthsOrder = Planet2.Earth.rawValue
    print(earthsOrder)
    // Prints: 0
    print(Planet2.Uranus.rawValue)
    // Prints: 13

    // Initializing from a raw value
    var currentPlanet2 = Planet2(rawValue: 1)

    print("\(currentPlanet2 ?? Planet2.Earth) is selected")
    // Prints:
    // Mars is selected

```

With *enum associated values* you can add additional details to your enumeration:

```

enum Action {
    case turnLeftDegree (Double),
        turnRightDegree (Double),
        makeForwardSteps (Int),
        makeBackwardSteps (Int),
        saySomething (String)
}

```

With this you can for example specify not only that you are going forward, but also how many steps ([Chapter 5: Tuples, switch and extensions](#), [Section 2: switch - case statement](#) for more details about *switch-case*):

```

var currentAction = Action.makeForwardSteps(10)

switch currentAction {
case .turnLeftDegree(let degree):
    print("Turn left by \(degree) degree")
case let .turnRightDegree(degree):
    print("Turn right by \(degree) degree")
case .makeForwardSteps(let steps):
    print("Make \(steps) step(s) forward")
case let .makeBackwardSteps(steps):
    print("Make \(steps) step(s) backward")
case let .saySomething(text):
    print("Say: \(text)")
}

```

```
// Prints: Make 10 step(s) forward
```

You may 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. If you look carefully into above code, you will notice that I put **let** in two different places. If you have more than one associated value, then some of them may be extracted as constants while other as variables. In such a case you have to put **var** or **let** prefixes before corresponding name (in brackets):

```
case .turnLeftDegree(let degree):
```

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:

```
case let .turnRightDegree(degree):
```

In this simple example both forms have the same effect.

Range operators

Swift introduces very handy *range operators*. You have seen an example of ranges in *Arrays concatenation and ranges usage* part of this chapter's *Section 1: Arrays*. Swift provides two *range operators* as a shortcut for expressing a range of values.

- The *closed range operator* `a...b` defines a range that runs from `a` to `b`, including both values. The value of `a` must not be greater than `b`.
- The *half-open range operator* `a..b` defines a range that runs from `a` to `b`, but does not include `b`. The value of `a` must not be greater than `b`. If the value of `a` is equal to `b`, then the resulting range will be empty.
- Starting from Swift 4 we can omit the upper or lower bound of a range specification to create a one-sided range.

```
let someArray = ["one", "two", "three", "four", "five"]
print(someArray[3...])
// Prints: ["four", "five"]

print(someArray[...2])
// Prints: ["one", "two", "three"]

print(someArray[..<2])
// Prints: ["one", "two"]

print(someArray[2..4])
// Prints: ["three", "four"]

// It's not correct
```

```
//print(someArray[2<..4])
```

Game code – add initializer

Complete initialization of `Board` class

At this moment you have a "stub" or skeleton of a `Board` class with two constants and one variable inside:

```
class Board {  
  private let rows, cols: Int  
  private var board: [[CellType]]  
}
```

If you have both lines uncommented, you will see two errors:

- Class 'Board' has no initializers
- Cannot find type 'CellType' in scope

Now, you are ready to fix them.

Add at the end of class, before closing curly bracket `}`, enum type:

```
enum CellType {  
  case none, empty, hit, notAllowed, rescue, ship, shot  
}
```

You define a new type `CellType` "hiding" under the names `none`, `empty`, `hit`, etc integer values. Saying the truth you pay greater attention to names than values.

The meaning of each case is as follows:

- `none` – nothing, none cell should have this value; use to signal unexpected problems;
- `empty` – empty cell; cell where you can put a ship or you can shot;
- `hit` – cell where you have already shot and hit;
- `notAllowed` – cell you can't put a ship or shot;
- `rescue` – cell around sunken ship;
- `ship` – part of a ship;
- `shot` – cell where you have already shot but miss.

Now you will add `initializer` where you will give values to all properties in your class (both constants and variables). Initializer is a class method with reserved name `init` (you can put this method just after enum `CellType`):

```
init(rows: Int = 10, cols: Int = 10) {  
    self.rows = rows  
    self.cols = cols  
  
    board = Array(repeating: Array(repeating: .none,  
                                   count: cols+2),  
                  count: rows+2)  
  
    prepareBoard()  
}
```

Notice:

- You use default values for `rows` nad `cols`.
- Defining board you add 2 to cols and rows – this is because a frame enclosing all game fields (see [Chapter 1: Initial steps, Section 1: Battleship game](#)).
- To have code clean, all other steps needed to initialize game board you "delegate" to `prepareBoard` method which you should implement as next to avoid this error: Cannot find 'prepareBoard' in scope.
- You have to use `self` keyword to differentiate property names (`self.rows`) and arguments (`rows`).

Every instance of a type has an implicit property called `self`, which is exactly equivalent to the instance itself. You use the `self` property to refer to the current instance within its own instance methods. It is the same as `this` in Java or C++. In practice, you don't need to write `self` in your code very often. If you don't explicitly write `self`, Swift assumes that you are referring to a property or method of the current instance whenever you use a known property or method name within a method. The main exception to this rule occurs when a parameter name for an instance method has the same name as a property of that instance. In this situation, the parameter name takes precedence, and it becomes necessary to refer to the property in a more qualified way. And that's why you use the `self` property to distinguish between the parameter name and the property name. See [SELF] for more details.

`prepareBoard` is quite self descriptive (put this method after `init` method):

```
func prepareBoard() {
    for i in 0...rows+1 {
        board[i][0] = .notAllowed
        board[i][cols+1] = .notAllowed
    }
}
```

```

for i in 0...cols+1 {
    board[0][i] = .notAllowed
    board[rows+1][i] = .notAllowed
}

for r in 1...rows {
    for c in 1...cols {
        board[r][c] = .empty
    }
}
}

```

At this step all errors are gone but nothing happens because you haven't instantiated **Board** class – there is no variables of **Board** type.

Just after **prepareBoard** add **printBoard** method intended to display game board (you will start implementing this method in next chapter):

```

func printBoard() {
}

```

Create **Engine** class

You don't want to manually managed all game object. Instead you will create a class which you will use as an **entry point to the game**. All interaction with a game is only through this class.

Create new file, name it **Engine.swift**, and paste the following code:

```

import Foundation

class Engine {
    private var boardPlayer: Board
    private var boardOpponent: Board

    init(rows: Int = 10, cols: Int = 10) {
        self.boardPlayer = Board(rows: rows, cols: cols)
        self.boardOpponent = Board(rows: rows, cols: cols)
    }

    func printBoards() {

```



```
        print("PLAYER")
        boardPlayer.printBoard()
        print("OPPONENT")
        boardOpponent.printBoard()
    }
}
```

There is nothing new to comment here.

Final step

The final step is to create **Engine** object and call its **printBoards** method. Paste the following code to `main.swift` file (comment all other code, being an effect of previous tests and experiments, you may have in this file):

```
import Foundation

let game = Engine(rows: 12, cols: 15)
game.printBoards()
```

You may now safely compile and run your code. You will see:

```
PLAYER
OPPONENT
Program ended with exit code: 0
```

which is not very spectacular result but, what is most important, fully correct result.

Code changes summary

Now class **Board** is fully initialized. You have also an initial version of **Engine** class.

Type methods, guards and string interpolation

You will do:

In this part you will start implement method printing game board.

You will learn:

- What **type methods** are.
- For what and how you can use **guards**.
- What a **string interpolation** is.

Type methods

In [Chapter 1: Initial steps](#), [Section 1: Game battleship](#) I presented some assumptions related to game you will implement – among others the way you will present game board to the player. To remind you game board, for example in case of 12 rows and 15 columns, should be printed as:

```

          111111
    123456789012345
+++++
1+. . . . . . . . . . +
2+. . . . . . . . . . +
3+. . . . . . . . . . +
4+. . . . . . . . . . +
5+. . . . . . . . . . +
6+. . . . . . . . . . +
7+. . . . . . . . . . +
8+. . . . . . . . . . +
9+. . . . . . . . . . +
10+. . . . . . . . . . +
11+. . . . . . . . . . +
12+. . . . . . . . . . +
+++++
```

where `+` represents `notAllowed` field and `.` empty field. What seems to be crucial for this printing is the way you align columns: you have to add a proper number of spaces before some text (I marked spaces with character `s` for aligning numbers, `#` to compensate frame existence and `*` to replace nonexistent digits at some positions):

```
ss#*****111111    12 spaces
ss#123456789012345  3 spaces
ss+++++            2 spaces
```

```

s1+.....+ 1 space
s2+.....+ 1 space
s3+.....+ 1 space
s4+.....+ 1 space
s5+.....+ 1 space
s6+.....+ 1 space
s7+.....+ 1 space
s8+.....+ 1 space
s9+.....+ 1 space
10+.....+ 0 spaces
11+.....+ 0 spaces
12+.....+ 0 spaces
+++++

```

To make it, you need a method determining the number of digits needed to print the largest row number so you could correctly compute the number of spaces printed in the place of **s** characters. Such a method, you may name it **determineNumberOfDigits**, is not a strict method of **Board** class – it is rather universal method which may be used by many other classes or methods. This is why you will put its code in separate class where you will "collect" all helper or useful method which don't belong to only one class. Create the **Utils** class with the frame of our method

```

class Utils {
    class func determineNumberOfDigits(number: Int) -> Int {
        [... PUT METHOD CODE HERE ...]
    }
}

```

This code looks almost familiar except **class** keyword proceedings function (method). All functions you implemented so far were an examples of *instance methods*. *Instance*, because to use them, you need an instance of a class. You have to create an object and then call an (instance) method on this object. Other words, this kind of methods *need object* because they *operate directly on the objects*. With **class** keyword proceedings function you define *type methods* that is a method which "*belongs*" rather to a *whole type* of objects than particular object (see also [TM:1]). In consequence, *no object of this type is needed* to use this method. The need to determine the number of digits may occur in

many different types and is not something typical for board, ship or any other battleship game object but rather for integer numbers, no matter where they are used. That is why separating code of this method in a versatile class collecting different utility methods seems to be reasonable.

Before you implement this method, analyze the following example where I try to justify existence of type methods. **Transformer** class is dedicated to preserve some values (in this case only one integer) and do some transformation on them (in this case only one, named **calculateDoubleValue**):

```
class Transformer {  
    var x = 5  
  
    func calculateDoubleValue() -> Int {  
        return x * 2  
    }  
}
```

To use it, you simply have to create its instance and then call a method on this instance:

```
var transformer = Transformer()  
print(transformer.x)  
print(transformer.calculateDoubleValue())
```

You may want to make this class more versatile, so you add another method:

```
func calculateDoubleValue(ofNumber: Int) -> Int {  
    return ofNumber * 2  
}
```

You can use it specifying any integer as:

```
print(transformer.calculateDoubleValue(ofNumber: 7))
```

It's not bad but a bit weird. To transform number 7 you have to create an object storing other values. You do this (create an object) only to be able to call method; you don't need any value stored in this object! As you can see, such a method of proceeding is not the most appropriate. It would be nice to have a method you can use without need of instantiating it. Happily this is what you have in Swift – you can use *type method* prepending your method with `class` keyword:

```
class func calculateDoubleValue(ofNumber: Int) -> Int {  
    return ofNumber * 2  
}
```

Now you can call it as:

```
print(Transformer.calculateDoubleValue(ofNumber: 7))
```

Note that it looks similarly, but now you use class name (starting with capital letter t) in front of `calculateDoubleValue(ofNumber:)` method.

Going back to our game code, at the first attempt you may implement `determineNumberOfDigits(number:)` method as:

```
class func determineNumberOfDigits(number: Int) -> Int {  
    var value = 10  
  
    if number > 0 {  
        for digits in 1...10 {  
            if (value > number) {  
                return digits  
            }  
            value *= 10  
        }  
    }  
    return 0  
}
```

This is not bad but also not in Swift style. You will change it in next section. Going bac to the main topic of this section: if you have

`determineNumberOfDigits` implemented as a *type method* you can call it this way (you may paste this snippet at the end of `main.swift` file; remember to delete it when you finish your test):

```
let d = Utils.determineNumberOfDigits(number: 54312)
print(d)
// Prints: 5
```

You simply put method name and specify "namespace" it belong to – a class name. It is much more natural then creating an object without intention to use it – only to be able to call method which do nothing with object:

```
let utils = Utils()
let leadingPadding = utils.determineNumberOfDigits(number:
rows)
// You don't need 'utils' any more but it will exist as long as
// this method will not finish
```

Code changes summary

You should have `Utils` class with `determineNumberOfDigits` type method inside. This method is almost done – you will do slight but important change in next section.

Guards

You may notice in `determineNumberOfDigits` method that the whole block of code:

```
for digits in 1...10 {  
    if (value > number) {  
        return digits  
    }  
    value *= 10  
}
```

is embraced within `if` only to prevent them being executed in case of required conditions are not fulfilled. Being more accurate: you don't want to prevent some code being executed but rather prevent the rest of method being executed. This is an important difference. With code:

```
func myFunction() {  
    // CODE PRECEDING CONDITION  
  
    if condition {  
        // SOME CONDITIONAL CODE  
    }  
  
    // CODE FOLLOWING CONDITION  
}
```

you will execute `CODE PRECEDING CONDITION`, **possibly** execute `SOME CONDITIONAL CODE` and **for sure** execute `CODE FOLLOWING CONDITION`. It may happen that both `SOME CONDITIONAL CODE` and `CODE FOLLOWING CONDITION` **require** some other condition to be

met and **it makes no sense to execute any of them in case of breaking this condition:**

```
func myFunction() {  
    // CODE PRECEDING CONDITION  
  
    if importantCondition {  
        if condition {  
            // SOME CONDITIONAL CODE  
        }  
  
        // CODE FOLLOWING CONDITION  
    }  
}
```

In this short snippet it looks acceptable but for longer code, maybe with more nested conditions of this type, you will get few level of code indentation and set of closing curly brackets which may make the code less readable. The main idea of **if**, similar to "outer" condition in above code, is to check if some, **strictly required**, conditions are met. If not, you should **immediately escape this function** as it may not be possible to execute subsequent statements. That is so important that we clearly "mark" such an important places in our code that in Swift you have special **guard** statement dedicated to check all necessary conditions and to be used in place of **ifs**. With **guard** code looks more natural and *let you keep the code that handles a violated requirement next to the requirement*.

```
func myFunction() {  
    // CODE PRECEDING CONDITION  
  
    guard importantCondition else {ESCAPE}  
  
    if condition {  
        // SOME CONDITIONAL CODE  
    }  
  
    // CODE FOLLOWING CONDITION  
}
```

guard, unlike conditional statement **if**, always is used with **else** part because Swift always needs to know what to do in case of condition failure. This action, denoted as **ESCAPE** in the above code, **must transfer control to exit the code block in which the guard statement appears**. If you want to make some action and further proceed with executing your function, you will get an error. Look into this code, where only **print** statement is used in case of condition failure:

```
func myFunction(x: Int) {  
    // CODE PRECEDING CONDITION  
    let y = 2 * x  
  
    guard x > 2 else {print("Aaaa!!!")}  
  
    if y < 6 {  
        // SOME CONDITIONAL CODE  
    }  
  
    // CODE FOLLOWING CONDITION  
}
```

You can't compile this code because Swift complains: 'guard' body must not fall through, consider using a 'return' or 'throw' to exit the scope. So you must exit the scope, and **ESCAPE** must be a set of statements exiting your function. In most cases it is simply **return**, but you may put something more "elaborated":

```
func myFunction(x: Int) {  
    // CODE PRECEDING CONDITION  
    let y = 2 * x  
  
    guard x > 2 else {  
        print("Aaaa")  
        return  
    }  
  
    if y < 6 {  
        // SOME CONDITIONAL CODE  
    }  
  
    // CODE FOLLOWING CONDITION  
}
```

Of course you may live without **guard** and replace it with **if**:

```
func myFunction(x: Int) {  
  // CODE PRECEDING CONDITION  
  let y = 2 * x  
  
  if !(x > 2) {print("Aaaa"); return}  
  
  if y < 6 {  
    // SOME CONDITIONAL CODE  
  }  
  
  // CODE FOLLOWING CONDITION  
}
```

Natural question is: *Do I really need 'guards' statement? Is this not some fanciful whim?* What can I say? Imagine a long code, with lots of **ifs**. Most of them a "typical" conditional statements branching your code – this is for what we use them. Some of them, maybe one or two, a crucial for execution – they check some strictly required condition. At the first sight it's hard to say which of them. You have to look into the code, check all blocks etc. So we use **guards** to clearly mark those important conditions. Using a **guard** statement for requirements **improves the readability of your code**, compared to doing the same check with an **if** statement. It lets you write the code that's typically executed without wrapping it in an **else** block. And, what was mentioned earlier, it lets you keep the code that handles a violated requirement next to the requirement.

Code changes summary

When you know what **guard** is, you can modify **determineNumberOfDigits** type method located in **Utils** class:

```
class func determineNumberOfDigits(number: Int) -> Int {  
    var value = 10  
  
    guard number > 0 else {return 0}  
  
    for digits in 1...10 {  
        if (value > number) {  
            return digits  
        }  
        value *= 10  
    }  
  
    return 0  
}
```

Modification is slight (**if** is replaced by **guard**) but makes the code closer to Swift style.

String interpolation

If you have a variable or constant of `String` type you can simply print it:

```
var message = "Hello World!"  
print(message)  
// Prints: Hello World!
```

Similarly, you can print an `Int` variable:

```
var x = 5  
print(x)  
// Prints: 5
```

Now you may ask, how to put together both `String` and `Int` so it could be printed? Assume, that your goal is to print:

Variable x has value 5

One possible solution is given below:

```
message = "Variable x has value " + String(5)
```

You have to build this string manually, concatenating string with integer previously "transformed" into string. If you want to create string with more part this could be troublesome and in special cases extremely unreadable.

You may dream to have possibility to write:

```
message = "Variable x has value x"
```

Notice that your intention is to treat first **x** as a character but second should be replaced by value stored in variable **x**. Of course in this case there is no way to distinguish when **x** should be **x** and when should be substituted. To indicate some part of a string as a *placeholder* (the part which should be replaced with the current value of constant or variable) Swift uses *string interpolation*. Wrap the name in parentheses (and) and escape it with a backslash \ before the opening parenthesis:

```
message = "Variable x has value \(x)"
print(message)
// Prints: Variable x has value 5
```

String interpolation is a way to construct a new **String** value from a mix of constants, variables, literals, and expressions by including their values inside a string literal:

```
let age = 12
message = "If you are \(age), you are \(age < 30 ? "young" :  
"middle-aged")"  
print(message)  
// Prints: If you are 12, you are young
```

In the example above, the value of **age** variable (number **12**) is inserted into a string literal in place of **\(age)**. The value of **age** is also part of a compound expression later in the string where *ternary conditional operator* is used.

The *ternary conditional operator* is a special operator with three parts, which takes the form **condition ? met : unmet**. It's a shortcut for evaluating one of two expressions based on whether **condition** is true or false. If **condition** is true, it evaluates **met** and returns its value; otherwise, it evaluates **unmet** and returns its value. Expression used in

`message` string returns either `young` or `middle-aged` string depending on `age` variable value. The ternary conditional operator is shorthand for the code below:

```
if condition {  
  // CONDITION MET  
} else {  
  // CONDITION UNMET  
}
```

Main difference between ternary conditional operator and `if` conditional statement is that the first returns value and can be used as a part of complex statement as it is given in the example with `age`. It is possible to write:

```
let category = age < 30 ? "young" : "middle-aged"
```

but you can't write:

```
let category = if condition {  
  // CONDITION MET  
} else {  
  // CONDITION UNMET  
}
```

Code changes summary

No changes in code – you will use knowledge presented here in subsequent sections.

Game code

PRINTING FIRST LINE

Equipped with `determineNumberOfDigits(number:)` function, you may start implement `printBoard` method:

```
func printBoard() {
    let leadingPadding = Utils.determineNumberOfDigits(number:
rows)
    var leadingPaddingString = ""
    var line = ""
    var number = 0

    for _ in 1...leadingPadding {
        leadingPaddingString += "s" // Replace with space
    }

    // BEGIN Print first line
    line = leadingPaddingString + "#" // Replace with space

    // Print tens digits
    for c in 1...cols {
        number = c/10
        if number == 0 {
            line += "*" // Replace with space
        } else {
            line += "\(number)"
        }
    }

    print(line)
    // END Print first line
}
```

At the beginning you define three variables and one constant:

- `leadingPadding` is the number of digits for the biggest row number.
- `leadingPaddingString` is a string of spaces repeated `leadingPadding` times.
- `line` you will use to build each line you want to print.
- `number` is a variable you will use to print columns and rows numbers.

Some places in the code are marked as:

```
// Replace with space
```

When you complete implementing this method remember to replace all mock characters (`s`, `#`, `*`) with space. Now you use all of them to make clear how many spaces you have and which part of your code is responsible for generating them. When executed, this code will print:

```
PLAYER
ss#*****111111
OPPONENT
ss#*****111111
```

You can compare this result with example given in [Section 1](#) of this chapter. As you can see, output:

```
ss#*****111111
```

agrees with a first line of an example shown there.

PRINTING SECOND LINE

Very similar code prints second line with unity digits. Paste the following code at the end of `printBoard` method:

```
// BEGIN Print second line
line = leadingPaddingString + "#" // Replace with space

// Print unity digits
for c in 1...cols {
    number = c%10
    line += "\(number)"
}

print(line)
// END Print second line
```

For a game board with 12 rows and 15 columns this will print:

```
ss#*****111111
ss#123456789012345
```

To save space I show only a part of output related to one game board (board both for player and opponent are printed identically).

Code changes summary

In this chapter you started to implement `printBoard` method. It is not complete yet, but you should see columns numbers printed correctly. You will complete it in next chapter.

Tuples, switch and extensions

You will do:

In this part you will finish implementing printing game board method. You will also create a class related to ships and implement one method to use with this type.

You will learn:

- What **tuple** is.
- How Swift turn switch-case statement into handy tool.
- How to separate your code with **extensions**.

Tuples

Tuple (pronounced /təpeɪl/, /tupeɪl/ or sometimes /tjupeɪl/) is a well known concept from script programming languages and something I always want to have in C. Of course you can live without it and mimic with for example arrays or dictionaries but tuple is more natural. Tuple group multiple values into a single compound value. The values within a tuple can be of any type and do not have to be of the same type as each other. Below you have an example of a simple tuple (pair in this case) of type `(Int, String)` – first element of this tuple is an integer while second is a string:

```
// Tuple of type (Int, String)
let warning = (123, "This is a critical warning")
var (currentMessageCode, currentMessageText) = warning

print ("Message text: " + currentMessageText)
// Prints: Message text: This is a critical warning
print ("Message text: \" + currentMessageText + "\"")
// Prints: This is a critical warning
```

As in many cases before, if you don't care about some element of a tuple, you can use underscore character in place of variables corresponding to tuple's element:

```
(_, currentMessageText) = warning
```

This is especially useful in case of complex tuples:

```
let x = (1, 2, 3, 4, 5, "a", "b", "c", (1, 2))
let (_, _, _, _, _, letter, _, _, _) = x
```

```
print(letter)
// Prints: a
```

You can also use index numbers starting at zero to get tuple's element:

```
print (x.5)
// Prints: a
```

To make your code more readable, you can name the individual elements in a tuple when the tuple is defined:

```
let alert = (messageCode: 456,
             messageText: "This is an alert")

print ("Message text: " + alert.messageText)
// Print: Message text: This is an alert
```

But you don't have to provide name for every individual elements:

```
let y = (1, 2, 3, compoundElement: (4, second: 5), 6, 7, 8)

print(y.3.1)
// Prints: 5

print(y.compoundElement.1)
// Prints: 5

print(y.compoundElement.second)
// Prints: 5

print(y.3.second)
// Prints: 5
```

Tuples are great for temporary usage. They are not suited to being use as a complex data structure persisting for a long time. In such a case *structures* and *classes* are better choice. You may find them useful when you want to return more than one value from a function:

```
func doSomething(  
  withInteger int: Int  
) -> (square: Int, double: Int) {  
  let square = int * int  
  let double = int + int  
  
  return (square: square, double: double)  
}  
  
let x = doSomething(withInteger: 3)  
print(x.0) // Prints: 9  
print(x.1) // Prints: 6  
  
print(x.square) // Prints: 9  
print(x.double) // Prints: 6
```


switch - case statement

NO IMPLICIT FALL THROUGH

I'm not going to explain the general idea behind `switch - case` statement (in short: `switch` statement or simply `switch`) as I suppose you have ever heard about it. Instead I want to show how Swift turn good old-fashion switch into very handy tool. Below there is Swift's switch:

```
let text = "one"

switch text {
case "one", "One":
    print("Case ONE")
case "two", "Two":
    print("Case TWO")
default:
    print("Default")
}
// Prints: Case ONE
```

What may catch your eye is lack of `breaks` which are needed in most programming languages to prevent from fall into next case. In Swift, if you fall into `case`, then only this `case`'s code is executed. If you want to use C-style fall through behavior a `fallthrough` keyword must be used. The `fallthrough` keyword causes code execution to move to the next `case` or `default` block on a case-by-case basis. Other words, this is not "global" behavior for all `cases` within a given `switch` but concerns only the `case` inside which `fallthrough` is used:

```
switch text {
```

```

case "one", "One":
    print("Case ONE")
    fallthrough
case "two", "Two":
    print("Case TWO")
default:
    print("Default")
}
// Prints:
// Case ONE
// Case TWO

```

Remember that doing that, the `fallthrough` does not check the case condition for the `switch` case that it causes execution to fall into. The `fallthrough` keyword simply causes code execution to move directly to the statements inside the next `case` (or `default` case) block without any case matching, as it is done in C:

```

var number = 2
switch number {
case 1, 2:
    print("1 or 2")
    fallthrough
case 3, 4:
    print("3 or 4")
default:
    print("all other options")
}
// Prints:
// 1 or 2
// 3 or 4

```

NO EMPTY CASES

As you saw, you can specify multiple values to match in one `case`:

```

case "one", "One":

```

and in Swift you can't leave empty `case` – a `case` without any instruction (which is typical for C-like code):

```

case "one":
case "One":
    print("Case ONE")

```


If you try to do this, you will see an error: 'case' label in a 'switch' should have at least one executable statement.

INTERVAL MATCHING

Another improvement in Swift is an ability of `switch`'s cases to check if their values are included in an interval:

```
let number = 12
switch number {
case 1...10:
    print("Range one")
case 11..<15:
    print("Range two")
case 15:
    print("Range three")
default:
    print("Out of range")
}
// Prints: Range two
```

TUPLE MATCHING

Also tuples can be tested by `case` statement which can be very handy and allows to simplify your code:

```
var point2D: (Double, Double)
point2D = (2.5, 2)
switch point2D {
case (0, 0):
    print("Origin")
case (_, 0):
    print("Point is on the OX axis")
case (0, _):
    print("Point is on the OY axis")
case (1..<2, 1..<2), (2...3, 2...3):
    print("Point is inside the restricted area")
default:
    print("Free 2D point")
}
// Prints: Point is inside the restricted area
```

BIND VALUE TO CONSTANTS OR VARIABLES

Other things which may be useful is the ability to bind the value a **switch** matches to temporary constants or variables, to be used in the body of the **case**:

```
point2D = (0, 5)
switch point2D {
case (0, 0):
    print("Origin")
case (let x, 0):
    print("Point \(x),0 is on the OX axis")
case (0, let y):
    print("Point (0, \(y)) is on the OY axis")
case let (x, y):
    print("Free 2D point \(x), \(y)")
}
// Prints: Point (0, 5.0) is on the OY axis
```

As it was mentioned in case of enumerations in [Chapter 3: Arrays and enumerations](#), [Section 2: Enumerations](#) you may put **let** or **var** in two different places. If you have more than one value, you may bind them to constants or variables. In such a case you have to put **var** or **let** prefixes before corresponding name:

```
case (let x, 0):
```

If all of the values for a **case** are binded as constants, or if all are binded as variables, you can place a single **var** or **let** annotation before the case name, for brevity:

```
case let (x, 0):
```

In this simple example both forms have the same effect.

COMPLEX MATCHING CONDITIONS

A switch case can use a where clause to express complex matching conditions:

```
point2D = (2, 3)
switch point2D {
```

```

case (0, 0):
    print("Origin")
case (let x, 0):
    print("Point \(x),0) is on the OX axis")
case (0, let y):
    print("Point (0, \(y)) is on the OY axis")
case let (x, y) where x > y:
    print("Point \(x), \(y) from a 2D subspace")
default:
    print("Eeee...")
}
// Prints: Eeee...

```


Extensions

Using type method like `determineNumberOfDigits(number:)` (see: [Chapter 4: Type methods, guards and string interpolation, Section 1: Type methods](#)) *to separate common code* which "belongs" rather to a whole type of objects than particular object is one possible option how you can solve this issue. If we pay a lot more attention to it, we discover that in this example determining a number of digits is something we do on particular integer object; it is something typical to integers. Saying the truth we made it as a type method for didactic reasons to describe what a type method is. Now we will show how this type on problems could be accomplish in more swifty style with *extensions*.

Extension add new functionality to an **existing** class, structure, enumeration, or protocol type. What is very important, this includes the ability to extend types for which we do not have access to the original source code. Extensions are declared with the `extension` keyword:

```
extension TypeYouExtend {  
    // New functionality to add to TypeYouExtend goes here  
}
```

Consider simple functionality related to integer numbers: you may want to have function constraining integer value to be within a given range. Such a function (`constrain(value, lowerEnd, upperEnd)`) is something very common to use in combination with `map(value, fromLow, fromHigh, toLow, toHigh)` while working with microcontrollers [ARD:1,2]. This is how you can implement it in Swift:

```

extension Int {
  func constrain(toRangeFrom min: Int, to max: Int) -> Int {
    if self > max {
      return max
    } else if self < min {
      return min
    }
    return self
  }
}

```

This way you add new method to an existing `Int` type. This method behaves as any other method made by `Int` class creators. Particularly, `self` represents the current instance of a given type – it is an object on which you call this method (for more informations about `self` see: [Chapter 3: Arrays and enumerations, Section 4: Implementing initializers](#)). Now you can call it:

```

var x = 12
x = x.constrain(toRangeFrom: 5, to: 10)
print(x)
// Prints: 10

```

With `self` and `extension` you can do even more. The following code is intended to calculate square of an integer:

```

extension Int {
  func square() -> Int {
    return self * self
  }
}

let x = 3
var y = x.square()

print(x)
// Prints: 3

print(y)
// Prints: 9

```

You can turn this code to mutate (change in-place) given integer calculating its square:

```

extension Int {
    func square() -> Int {
        return self * self
    }

    mutating func squareMe() {
        self = self * self
    }
}

let x = 3
var y = x.square()
print(x)
// Prints: 3

print(y)
// Prints: 9

y.squareMe()
print(y)
// Prints: 81

```

Now you can use this concept to implement another helpful method. This method should enlarge specified string to a given length left padding it with spaces by default or any other character if specified. For example, if string **12** should be transformed into four-character string, this method should return **..12** where dots **.** are used in a place of spaces to make it visible. Create the `Extensions.swift` file and put inside the following code:

```

extension String {
    func leftPadding(
        toLength: Int,
        withPad: String = " "
    ) -> String {
        guard toLength > self.count else {return self}

        let padding = String(repeating: withPad,
                               count: toLength - self.count)

        return padding + self
    }
}

```

```
}
```

You can test this extension placing for a while the following code in `main.swift` file:

```
let x = 12
let s1 = String(x).leftPadding(toLength: 4, withPad: "*")
let s2 = "\\(x)".leftPadding(toLength: 4, withPad: "#")

print(s1)
// Prints: **12

print(s2)
// Prints: ##12
```

Notice how you create `padding` string. In `printBoard` function a variable `leadingPaddingString` is created as `leadingPadding` spaces concatenated together in `for` loop:

```
for _ in 1...leadingPadding {
    leadingPaddingString += " "
}
```

More swifty way is to use initializer with `repeating` argument. You have seen this in previous chapter in `Board` initializer where two-dimensional array was created:

```
board = Array(repeating: Array(repeating: .none,
                                count: cols+2),
              count: rows+2)
```

Because we extend functionality of `String` class, `self` refers to a given string object. So `self.count` is about the number of characters in it, while `padding + self` is a concatenation of sequence of spaces (or other character specified as `withPad` argument) and string itself (`padding` and `self` part respectively).

Complete game board printing method

Now you are ready to complete last part of a game board printing method. The rest of the `printBoard` method code shouldn't be difficult to understand (paste this code at the end of `printBoard` before closing curly bracket `}`):

```
// BEGIN Print all gameboard rows
for r in 0...rows+1 {
    line = ""

    if r == 0 || r == rows+1 {
        line += leadingPaddingString
    } else {
        line += String(r).leftPadding(toLength: leadingPadding,
                                     withPad: "s")
        // Replace with space
    }

    for c in 0...cols+1 {
        switch board[r][c] {
            case .empty:
                line += "."
            case .hit:
                line += "!"
            case .ship:
                line += "X"
            case .shot:
                line += "*"
            case .none:
                line += "?"
            case .notAllowed:
                line += "+"
            case .rescue:
                line += "0"
        }
    }

    print(line)
}
// END Print all game board rows
```

For a game board with 12 rows and 15 columns this will print:

```
ss#*****111111
ss#123456789012345
ss+++++++
s1+.....+
s2+.....+
s3+.....+
s4+.....+
s5+.....+
s6+.....+
s7+.....+
s8+.....+
s9+.....+
10+.....+
11+.....+
12+.....+
ss+++++++
```

To save space I show only a part of output related to one game board (board both for player and opponent are printed identically).

Finally you can search for every places marked in your code as **Replace with space** and replace **s**, **#** and ***** with space character to get final result:

```
      111111
    123456789012345
  ++++++
1+.....+
2+.....+
3+.....+
4+.....+
5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
11+.....+
12+.....+
  ++++++
```


Game code

Add `ship` class

Three elements are needed to place a ship:

- **size** so you know how many successive cells the ship occupies;
- **coordinates** of the first element so you know when it starts (I will use *anchor* or *reference point* for this);
- **direction** so you know how the ship is oriented: upwards, downwards, leftwards or rightwards.

For the ship of size 3 you have:

```
      start column
      |
      |
.....
....u....
....u....
..llXrr..---start row
....d....
....d....
.....
```

where

- X – first element; it has (start row, start column) coordinates;

- u – successive cells the ship occupies if it is directed upwards;
- d – successive cells the ship occupies if it is directed downwards;
- l – successive cells the ship occupies if it is directed leftwards;
- r – successive cells the ship occupies if it is directed rightwards.

Create a new class, as you did it before, and name it **Ship**. As for now, this class will have only one component: **Direction** enumeration used to uniquely identify or position a ship on a game board:

```
class Ship {
    enum Direction {
        case up, down, left, right
    }
}
```

mayPlaceShip function, part 1 of 2

mayPlaceShip function is intended to check if it is possible to put a ship of a given **size**, starting a **anchor** position and directed towards **direction**. Put the following code at the end of **Board** class:

```
func mayPlaceShip(
    size: Int,
    anchor: (row: Int, col: Int),
    direction: Ship.Direction
) -> Bool {
    var modifier = (forRow: 0, forCol: 0)
    var r = 0
    var c = 0

    switch direction {
    case .up:
        modifier = (forRow: -1, forCol: 0)
    case .down:
        modifier = (forRow: +1, forCol: 0)
    case .left:
        modifier = (forRow: 0, forCol: -1)
    case .right:
        modifier = (forRow: 0, forCol: +1)
```

```

}

for i in 0...size-1 {
    r = anchor.row + i * modifier.forRow
    c = anchor.col + i * modifier.forCol

    guard r>0, r<rows+1 else {return false}
    guard c>0, c<cols+1 else {return false}

    if board[r][c] != .empty {
        return false
    }
}

return true
}

```

This is not a final version of this method; you will update it soon. In this form you should have no problems to understand how it works.

Create `Test` class

Create `Test` class with the following `testMayPlaceShip` type method:

```

class Test {
    // For 10 x 10 game board should prints:
    // possible, impossible, possible, impossible
    class func testMayPlaceShip(board: Board) {
        var x = board.mayPlaceShip(size: 4,
                                    anchor: (row: 2, col: 2),
                                    direction: .down)

        print(x ? "possible" : "impossible")

        x = board.mayPlaceShip(size: 4,
                                anchor: (row: 9, col: 2),
                                direction: .down)

        print(x ? "possible" : "impossible")

        x = board.mayPlaceShip(size: 4,
                                anchor: (row: 5, col: 7),
                                direction: .right)

        print(x ? "possible" : "impossible")
    }
}

```

```

        x = board.mayPlaceShip(size: 4,
                                anchor: (row: 5, col: 8),
                                direction: .right)

        print(x ? "possible" : "impossible")
    }
}

```

Depending on your game board size this method prints different results; for 10 by 10 game board you will see: possible, impossible, possible, impossible.

Next add this `test` method at the end of `Engine` class:

```

func test() {
    Test.testMayPlaceShip(board: boardPlayer)
}

```

Finally add this line at the end of `main.swift` file and run your code:

```
game.test()
```

Be sure to have 10 by 10 game board – you should have only the following code in `main.swift` file:

```

import Foundation

let game = Engine(rows: 10, cols: 10)
game.printBoards()
game.test()

```

In **Console** window you will see:

```

PLAYER
      1
    1234567890
+++++
1+.+.+.+.+.+.+.
2+.+.+.+.+.+.+.
3+.+.+.+.+.+.+.
4+.+.+.+.+.+.+.
5+.+.+.+.+.+.+.
6+.+.+.+.+.+.+.

```



```

7+.....+
8+.....+
9+.....+
10+.....+
+++++++
OPPONENT
      1
    1234567890
+++++++
1+.....+
2+.....+
3+.....+
4+.....+
5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
+++++++
possible
impossible
possible
impossible
Program ended with exit code: 0

```

Last four lines (excluding exit status code line) are result of your `test` method – you will see there: possible, impossible, possible, impossible.

Code changes summary

In this chapter you:

- add an `Extensions` class with an extension for strings (`leftPadding` function);
- finished `printBoard` from `Board` class;
- created `Ship` class;

- add `mayPlaceShip` method to `Board` class (you will complete this method soon);
- add `Test` class to keep together all test methods.

Now you can print game board and check if some location is a good place to put there a ship.

Properties, dictionaries and sets

You will do:

In this part you will finish implementing printing game board method. You will also create a class related to ships and implement one method to use with this type.

You will learn:

- Various **property types**.
- How to create and use **dictionaries**.
- How to create and use **sets**.

Property types

Variables or constants present inside a class are called *properties* of this class. Being more precisely, properties of objects which are instances of this class. In Swift, besides "normal" properties, called *stored properties*, you have worked so far, there are also few other types of properties.

Computed properties

Classes, structures and enumerations can define *computed properties*. This type of properties do not actually store a value. Instead, they provide a *getter* (programmers jargon term for `get` method) and an *optional setter* (programmers jargon term for `set` method) to retrieve and set this and other properties indirectly. So computed properties, in contrast to stored properties, calculate rather than store value. Simply speaking, sometimes you don't need to store explicitly (permanently) some value – for example it may be too expensive (taking into consideration memory usage) or this value may be computed based on other values.

To define computed property you use `get` and `set` keyword. In case you want to have a read only computed property you can skip the `get` keyword (and of course there must be no `set` because this is read only property) or even `return` if the entire body of the function is a single expression (see also [Chapter 2: Variables and functions](#), [Section 3: Functions](#), subsection: [Function with an implicit return](#)).

To justify existence of computed properties consider a following example of **Square** class:

```
class Square {
    var edge = 2.0

    func getArea() -> Double {
        return edge * edge
    }
}

var s = Square()
print("\(s.edge) \(s.getArea())")
// Prints:
// 2.0 4.0
s.edge = 5
print("\(s.edge) \(s.getArea())")
// Prints:
// 5.0 25.0
```

This is a quite simple class representing a square. It has only one property: **edge**, and one method: **getArea()**. Defining square by its edge is one of possible options. Other is providing area and then calculate edge length:

```
class Square {
    var area = 4.0

    func getEdge() -> Double {
        area.squareRoot()
    }
}

var s = Square()
print("\(s.getEdge()) \(s.area)")
// Prints:
// 2.0 4.0

s.area = 25
print("\(s.getEdge()) \(s.area)")
// Prints:
// 5.0 25.0
```


As you can see, it's quite natural to consider both *edge* and *area* as a property of a square; it's quite natural to have an access of the form: **s.edge** and **s.area**. So another approach to implement this might take a form:

```
class Square {
    var edge: Double
    var area: Double

    init(edge: Double) {
        self.edge = edge
        area = edge * edge
    }
}
```

This looks good as long as you only use values from instantiated square:

```
var s = Square(edge: 2)
print("\(s.edge) \(s.area)")
// Prints:
// 2.0 4.0
```

Unfortunately it will fail if you try to change some of square's properties:

```
s.edge = 5
print("\(s.edge) \(s.area)")
// Prints:
// 5.0 4.0
```

It would be nice to combine property with method and have *computable* property:

```
class Square {
    var edge = 2.0
    var area: Double {
        edge * edge
    }
}

var s = Square()
print("\(s.edge) \(s.area)")
// Prints:
```

```
// 2.0 4.0

s.edge = 5
print("\(s.edge) \((s.area)")
// Prints:
// 5.0 25.0
```

This fixes **Square** class in "one direction" – get direction, because if you try to set properties' values, you will get an error:

```
// Error:
// Cannot assign to property: 'area' is a get-only property
s.area = 36
```

The final version is given below:

```
class Square {
    var edge = 2.0
    var area: Double {
        get {
            edge * edge
        }
        set (newValue) {
            edge = newValue.squareRoot()
        }
    }
}

var s = Square()
print("\(s.edge) \((s.area)")
// Prints:
// 2.0 4.0

s.edge = 5
print("\(s.edge) \((s.area)")
// Prints:
// 5.0 25.0

s.area = 36
print("\(s.edge) \((s.area)")
// Prints:
// 6.0 36.0
```

It was just a kind of mental experiment and I hope it has convinced you that computed properties might be useful. Of course, you can live without them, but their existence simplifies code and make it more

natural. Without them you would be forced to close an access to properties with `private` keyword in front of them and use only accessors methods, getters and setters. This is a typical pattern in Java: keep all properties hidden and use methods to have an access. For me, this is really annoying.

At the end of this section, one more example:

```
class ComputedPropertyTest {
    var simpleProperty = 3
    var computedProperty: Int {
        get {
            return simpleProperty * 2
        }

        set (newValue) {
            simpleProperty = newValue / 2
        }
    }
    var readOnlyComputedProperty: Int {
        simpleProperty * 3
    }
}

var cpt = ComputedPropertyTest()
var test = (sp: cpt.simpleProperty,
            cp: cpt.computedProperty,
            rocp: cpt.readOnlyComputedProperty)
print("\n(test.sp) \n(test.cp) \n(test.rocp)")
// Prints: 3 6 9

cpt.computedProperty = 8
test = (sp: cpt.simpleProperty,
        cp: cpt.computedProperty,
        rocp: cpt.readOnlyComputedProperty)
print("\n(test.sp) \n(test.cp) \n(test.rocp)")
// Prints: 4 8 12
```

Lazy stored properties

Another great concept in Swift is a *lazy stored property*. A lazy stored property is a property whose **initial value is not calculated until the first time it is used**. Lazy properties may be useful when the

initial value for a property is not known until after an instance's initialization is complete, for example we may use it for time consuming initialization process. Lazy property must always be declared as a variable (constant properties must always have a value before initialization completes).

```
class Action {
    init() {
        print("Init Action class") // 1
    }

    func doSomething(){
        print("Do something for Action instance") // 2
    }
}

class LazyAction {
    init() {
        print("Init LazyAction class") // 3
    }

    func doSomething(){
        print("Do something for LazyAction instance") // 4
    }
}

class ActionExecutor {
    var action = Action()
    lazy var lazyAction = LazyAction()

    func executeAction(){
        action.doSomething() // 5
        lazyAction.doSomething() // 6
    }
}

let ae = ActionExecutor() // 7
ae.executeAction() // 8
```

Here is what happens when you execute this code:

1. Statement marked with comment `// 7` is executed. In effect, a new instance of `ActionExecutor` will be created and reference to this object saved in `ae` variable.

2. In consequence you will see:

Init Action class

which is an effect of `print` function executed in `Action`'s initializer marked with `// 1` comment. Notice that `LazyAction`'s initializer wasn't executed as there is no message `Init LazyAction class` printed on screen. So now object referenced by `ae` is ready to use but lazy `lazyAction` property initialization was deferred to the moment you first time decide to use it (step 5 below).

3. Next, in line marked with comment `// 8` you call `executeAction()` method on `ae` object.
4. In the body of `executeAction()` first you call `doSomething()` function on `action` object (line marked with `// 5` comment). `Actions`'s `doSomething()` function prints a message (line marked with `// 2` comment):

Do something for Action instance

5. Next you call `doSomething()` function on `lazyAction` object (line marked with `// 6` comment). Saying the truth, you only try to make this call as object referenced by `lazyAction` doesn't exist yet. So before call will take effect, first an instance of `LazyAction` must be created. And this is why now you will see:

Init LazyAction class

which is an effect of `print` function marked by `// 3` comment in `LazyAction`'s initializer.

6. Finally, because now `lazyAction` object exist, function `doSomething()` can be executed resulting with output (line marked with `// 4` comment):

Do something for LazyAction instance

I intentionally use `LazyAction` as a name for a class to attract your attention. Please remember: what is lazy is property, not class.

NOTE

If a property marked as a `lazy` is accessed by multiple threads simultaneously and the property has not yet been initialized, there is no guarantee that the property will be initialized only once.

Property observers

Property observers are really great things. They monitor to any changes in a property's value. Every time a property's value is set, even if the new value is not really new, observer is called. You can add property observer to any stored properties, except for lazy properties.

There are two observers:

- `willSet` is called just **before** the value is stored. This observer gets new property value as a constant parameter with a default name of `newValue`. You can specify your own name if you don't like this one.
- `didSet` is called just **after** the new value is stored. This observer gets old (previous)) property value as a constant parameter with a default name of `oldValue`. You can specify your own name if you don't like this one.

In example below observers for `secondPropertyToObserve` defines their own names: `newValueToBeSet` and `oldValueToBeReplaced` instead of default `newValue` and `oldValue` used in `firstPropertyToObserve`:

```
class ClassWithPropertyObservers {
    var firstPropertyToObserve: Int = 3 {
        willSet {
            print("firstPropertyToObserve is going to get new value
of: \(newValue)")
        }
        didSet {
            print("An old value of firstPropertyToObserve (\
(oldValue)) has just been replaced by a new one")
        }
    }

    var secondPropertyToObserve: Int = 7 {
        willSet(newValueToBeSet) {
            print("secondPropertyToObserve is going to get new value
of: \(newValueToBeSet)")
        }
        didSet(oldValueToBeReplaced) {
            print("An old value of secondPropertyToObserve (\
(oldValueToBeReplaced)) has just been replaced by a new one")
        }
    }
}

var cwpo = ClassWithPropertyObservers()
cwpo.firstPropertyToObserve = 5
// Prints:
// firstPropertyToObserve is going to get new value of: 5
// An old value of firstPropertyToObserve (3) has just been
// replaced by a new one

cwpo.secondPropertyToObserve = 9
// Prints:
// secondPropertyToObserve is going to get new value of: 9
// An old value of secondPropertyToObserve (7) has just been
// replaced by a new one
```

NOTE

When a default value is assigned to a stored property, or its initial value is set with an initializer, the value of that property is set directly, without calling any property observers.

Type properties

Type properties are properties related with particular type rather than particular instance of that type. They are like static variables or constants in C or Java.

```
class ClassWithTypeProperty {
    static var storedTypeProperty = 22
    var computedProperty: Int {
        return ClassWithTypeProperty.storedTypeProperty * 3
    }
}

print("\(ClassWithTypeProperty.storedTypeProperty)")
// Prints: 22

ClassWithTypeProperty.storedTypeProperty = 222

print("\(ClassWithTypeProperty.storedTypeProperty)")
// Prints: 222

let obj1 = ClassWithTypeProperty()
let obj2 = ClassWithTypeProperty()

print("\(obj1.computedProperty), \(obj2.computedProperty)")
// Prints: 666, 666
```

Frequently type properties are used as a "counters" or "singletons" for all instances of a given class:

```
class A {
    var v = 0 {
        willSet {
            A.sum += newValue
        }
    }
}
```



```

    static var sum = 0
}

var obj1 = A()
var obj2 = A()
var obj3 = A()

// Error:
// Static member 'sum' cannot be used on instance of type 'A'
// print("\n(obj1.v) \n(obj2.v) \n(obj3.v) \n(A.sum)")

print("\n(obj1.v) \n(obj2.v) \n(obj3.v) \n(A.sum)")
// Prints:
// 0 0 0 0

obj1.v = 5
obj2.v = 7
obj3.v = -4
obj2.v = 2

print("\n(obj1.v) \n(obj2.v) \n(obj3.v) \n(A.sum)")
// Prints:
// 5 2 -4 10

```

Property wrappers

Discussing this topic is beyond the scope of this book. If you are curious, please read for example [SD:1, SD:2].

Dictionaries

A dictionary stores associations between keys (all of the same type) and values (all of the same but possible different than keys type) with no defined order. As a key you can use any hashable type

You define dictionary quite similar as you do for arrays:

```
var dictionary1 = [String:String]() // shorthand form
var dictionary2 = Dictionary<String, String>() // full form
var dictionary3 = ["digit0": "zero",
                  "digit1": "one",
                  "digit2": "two"]

print(dictionary1)
// Prints: [:]
print(dictionary2)
// Prints: [:]
print(dictionary3)
// Prints:
// ["digit2": "two", "digit0": "zero", "digit1": "one"]
```

You can create dictionaries from a sequence of key-value pairs:

```
let streamOfTuples = [(1, "one"),
                     (2, "two"),
                     (3, "three"),
                     (4, "four")]
let dict01 = Dictionary(
    uniqueKeysWithValues: streamOfTuples)
print(dict01)
// Prints:
// [3: "three", 4: "four", 1: "one", 2: "two"]
```

This can be completed even faster with `zip(_:_)` function:

```

let words = ["one", "two", "three", "four"]
let dict02 = Dictionary(uniqueKeysWithValues: zip(
    1...,
    words)
)
print(dict02)
// Prints:
// [1: "one", 3: "three", 4: "four", 2: "two"]

```

You can easily combine two dictionaries into one. This is almost as easy as for arrays but you cannot use simply `+` operator because you must resolve somehow conflicts caused by duplicate keys. Instead use `merge(_:uniquingKeysWith:)` method where you specify a closure to deal with conflicts:

```

var d1 = [1: "1a", 2: "1b", 3: "1c"]
let d2 = [2: "2b", 3: "2c", 4: "2d"]

d1.merge(d2){(v1, v2) in v2}
print(d1)
// Prints:
// [4: "2d", 3: "2c", 1: "1a", 2: "2b"]

```

Dictionaries have a fancy initializer `Dictionary(grouping:by:)`, and its job is to convert a sequence into a dictionary based on any grouping you want:

```

let words = ["aa", "ab", "ac", "ba", "bb", "bc"]
let d = Dictionary(grouping: words){ $0.first! }
print(d)
// Prints:
// ["b": ["ba", "bb", "bc"], "a": ["aa", "ab", "ac"]]

```

Dictionary usage is not much different from array usage:

```

var dictionary3 = ["digit0": "zero", "digit1": "one", "digit2": "two"]
dictionary3["digit2"] = "Tw0"

print(dictionary3)
// Prints:
// ["digit2": "Tw0", "digit0": "zero", "digit1": "one"]
var oldValue = dictionary3.updateValue("TW0", forKey: "digit2")
print(dictionary3)

```

```

// Prints:
// ["digit2": "Tw0", "digit0": "zero", "digit1": "one"]
print("Old value: \(oldValue ?? "no value")")
// Prints: Old value: Tw0

dictionary3["digit3"] = "three"
print(dictionary3)
// Prints:
// ["digit3": "three", "digit2": "Tw0", "digit0": "zero",
"digit1": "one"]

for (key, value) in dictionary3 {
    print("key: \(key) value: \(value)")
}
// Prints:
// key: digit3 value: three
// key: digit2 value: Tw0
// key: digit0 value: zero
// key: digit1 value: one

for key in dictionary3.keys {
    print("key: \(key)")
}
// Prints:
// key: digit3
// key: digit2
// key: digit0
// key: digit1

for value in dictionary3.values {
    print("value: \(value)")
}
// Prints:
// value: three
// value: Tw0
// value: zero
// value: one

```

You can easily pick-up elements according to filter criteria. You provide a closure taking the key and value for each element, and any dictionary key-value pair you return true for is included in a resulting dictionary:

```

let all = [1: 2, 2: 4, 3: 3, 4: 5]
let filtered = all.filter {
    key, value in
    return (key + value) % 2 == 0
}
print(filtered)
// Prints:

```

```
// [3: 3, 2: 4]
```

.

Sets

A set stores **distinct** values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of elements is not relevant, or when you need to ensure that an element appears only once. A type of objects stored in a set must be hashable.

```
var setOfDigitsNames1 = Set<String>()
setOfDigitsNames1.insert("one")
setOfDigitsNames1.insert("two")

var setOfDigitsNames2: Set<String> = ["one", "two"]
var setOfDigitsNames3: Set = ["one", "two"]

print(setOfDigitsNames1)
// Prints: ["one", "two"]
print(setOfDigitsNames2)
// Prints: ["one", "two"]
print(setOfDigitsNames3)
// Prints: ["one", "two"]
```

Actions you can perform on sets are typical for sets: you can check if elements is in set, find an intersection of two sets (elements which belong to both sets), their union ("sum" – which is a result of combining both sets and removing duplicates) etc.:

```
if !setOfDigitsNames3.isEmpty {
    if setOfDigitsNames3.contains("two") {
        setOfDigitsNames3.remove("two")
    }
    if !setOfDigitsNames3.contains("three") {
        setOfDigitsNames3.insert("three")
    }
}
```

```

print(setOfDigitsNames3)
// Prints: ["one", "three"]

var setOfInts1: Set = [1, 2, 3, 4]
var setOfInts2: Set = [3, 4, 5, 6]

print(setOfInts1)
// Prints: [1, 4, 2, 3]
print(setOfInts2)
// Prints: [3, 4, 6, 5]

print(setOfInts1.union(setOfInts2))
// Prints: [1, 4, 6, 5, 2, 3]
print(setOfInts1.intersection(setOfInts2))
// Prints: [3, 4]
print(setOfInts1.subtracting(setOfInts2))
// Prints: [1, 2]
print(setOfInts1.symmetricDifference(setOfInts2))
// Prints:[1, 5, 2, 6]

```

There are few methods you can use to test set membership or equality:

1. To test if two sets contain exactly the same values, you use `==` operator.
2. To test if all of the values of a set are contained in the specified set, you use `isSubset(of:)` method. You use `isStrictSubset(of:)` if you want to exclude equality of both sets.
3. To test if a set contains all of the values from a specified set, you use `isSuperset(of:)` method. You use `isStrictSuperset(of:)` if you want to exclude equality of both sets.
4. To test if both sets have no common values, you use `isDisjoint(with:)` method.

Sets are not so widely used as arrays or dictionaries. Personally I use sets to simplify `ifs` syntax. Imagine you have an enumeration to mark system messages as warnings, errors or simply informations:

```
enum Message {
    case warning, error, info
}

var msg = Message.warning
```

Now you may filter messages according to some rules:

```
if msg == .warning || msg == .error {
    print("!!!")
}
// Prints:
// !!!
```

Instead of listing all of them in condition, which may results a clumsy and messy statement I use set as it is showed below:

```
// You can write:
// let importantMessages: Set = [Message.warning,
// Message.error]
// or use shorter form:
let importantMessages: Set<Message> = [.warning, .error]

if importantMessages.contains(msg) {
    print("!!!")
}
// Prints:
// !!!
```

As you can see with set condition simplifies to:

```
importantMessages.contains(msg)
```


Game code

Code changes related to `Ship` class

Extend previously created `Ship` class with new enumeration type (put this code before class's closing curly bracket `}`):

```
enum Status {  
    case damaged, destroyed, ready  
}
```

Status means:

- `ready` – ready to use, fully operational part of a ship.
- `damaged` – damaged part of a ship. In our implementation you will set ship segment's state to this value after it is hit. Potentially this state is reversible – you may implement method to recover it's state to fully operational. If all ship's segments are marked as damaged then ship is considered as destroyed and immediately all segments takes state `destroyed`.
- `destroyed` – is used in case of critical, not recoverable damages.

Add also to this class a set of properties along with initializer:

```
private let size: Int  
private var readyLevel: Int  
var position: [(row: Int, col: Int, status: Status)]
```

```

var isDestroyed: Bool {
    return readyLevel == 0 ? true : false
}

init(size: Int,
    position: [(row: Int, col: Int, status: Status)]) {
    {
        self.size = size
        self.position = position
        self.readyLevel = size
    }
}

```

- **size** is a size of a ship in terms of cells (segments) occupied by this ship (we allow only "straight" or "line aligned" ships where all ship's segments are in one line; bends are not allowed).
- **position** is an array of tuples describing each ship's segment: its location (**row** and **column**) and condition (**status**).
- **readyLevel** describe combat readiness and as for now will expressed the number of ship's ready segments.
- **isDestroyed** is true if the ship is destroyed (all segments marked as damaged and in consequence as destroyed) and may not longer take part in the fight. This property is an example of *computed property*.

Next add new methods:

```

func isLocatedAt(row: Int, col: Int) -> Bool {
    for coordinate in position {
        if coordinate.row == row &&
            coordinate.col == col {
            return true
        }
    }

    return false
}

func hitAt(row: Int, col: Int) {
    for (index, coordinate) in position.enumerated() {

```

```

        if coordinate.row == row &&
            coordinate.col == col {
            position[index].status = .damaged
            readyLevel -= 1
            break
        }
    }
}

```

`hitAt(row:col)` is intended to set state of ship segment and decrease its readiness, while `isLocatedAt(row:col)` checks if among ship segments exists one of coordinates `(row, col)`.

Add Ships class

`Ships` class collects informations about all ships belonging to one of the players. To do so, you create a class with the needed fields:

```

class Ships {
    var ships = [String: Ship]()
    var shipsAtCommand: Int {
        var shipsNumber = ships.count

        for (_, ship) in ships {
            if ship.isDestroyed {
                shipsNumber -= 1
            }
        }

        return shipsNumber
    }
}

```

As you can see, variable `ships` is defined as a *dictionary*. Iterating over this set returns a tuple (in our case) of the form:

(key: `String`, value: `Ship`)

In consequence, you may write the iteration code either in the form:

```

for element in ships {
    if element.value.isDestroyed {

```

```

        shipsNumber -= 1
    }
}

```

or, as you did, in the form:

```

for (key, value) in ships {
    if value.isDestroyed {
        shipsNumber -= 1
    }
}

```

Because in your case you know that **value** is a **Ship** object, so you use name **ship** instead to make code more readable. The first tuple's element, **key** is not needed in your code, what is singled by Xcode with the message `Immutable value 'key' was never used;` consider replacing with `'_'` or removing it. To silent this warning, **key** is finally replaced by underscore character `_` which is the way you say to Swift: *Swift, I know that there is something here, but I don't need it and so I don't care about it.*

Modifications in **Board** class

To keep relation between board and ships on that board introduce at the top of **Board** class new property:

```
var ships: Ships
```

This change require to modify an initializer. Add this line to `init()` before `prepareBoard()` call:

```
ships = Ships()
```

Based on previously created method

`mayPlaceShip(size:anchor:direction:)` add a new method:

```
func placeShip(
```



```

size: Int,
anchor: (row: Int, col: Int),
direction: Ship.Direction
) {
    var modifier = (forRow: 0, forCol: 0)
    var r = 0
    var c = 0
    var position = [(row: Int, col: Int, status: Ship.Status)]()

    switch direction {
    case .up:
        modifier = (forRow: -1, forCol: 0)
    case .down:
        modifier = (forRow: +1, forCol: 0)
    case .left:
        modifier = (forRow: 0, forCol: -1)
    case .right:
        modifier = (forRow: 0, forCol: +1)
    }

    for i in 0...size-1 {
        r = anchor.row + i*modifier.forRow
        c = anchor.col + i*modifier.forCol

        board[r][c] = CellType.ship
        position.append((row: r,
                        col: c,
                        status: Ship.Status.ready))
    }

    ships.ships["\(anchor)"] =
        Ship(size: size, position: position)
}

```

This code doesn't introduce any new elements. Variable **position** consist of iteratively added ship's parts which is done in:

```

for i in 0...size-1 {
    ...
    position.append((row: r,
                    col: c,
                    status: Ship.Status.ready))
}

```

You may stop for a while on the line:

```
ships.ships["\(anchor)"] = ship
```

`ships` is a dictionary with keys of the `String` type and `Ship` as a value type. Key should uniquely identify every ship. Combining each ship anchor coordinates (its first cell's row and column coordinate) you get a unique ship key because no more than one ship can start in a given cell (anchor). Such a string may be simply accomplish with "`\(anchor)`" phrase.

This method assumes that ship placement is possible, so remember to call `mayPlaceShip(size:anchor:direction:)` before.

There is one simplification, still also present in both methods: you don't care about cells surrounding the ship. Thus, as for now, two ships may be placed so they "touch" – this will be fixed in next chapter.

Modifications in `Engine` class

In `Engine` class add enumeration type to distinguish current player:

```
enum Who {  
    case player, opponent  
}
```

One of them is called *player* and the other is called (his/her) *opponent*. In practice, *player* can be identified with *human player*, while *opponent* with a *computer player*.

Having two different boards, you need a method to easily "switch" between them (or select correctly those related to a given player):

```
func getWhoBoard(who: Who) -> Board {  
    if who == Who.player {  
        return boardPlayer  
    }  
  
    return boardOpponent  
}
```

Finally, add two "boilerplate" methods which main purpose is to call, previously created in **Board** class, methods with correct arguments (that is board and ships dedicated to given player):

```
func mayPlaceShip(
    who: Who,
    size: Int,
    anchorRow: Int,
    anchorCol: Int,
    direction: Ship.Direction
) -> Bool {

    let anchor = (row: anchorRow, col: anchorCol)
    let boardWho = getWhoBoard(who: who)
    return boardWho.mayPlaceShip(size: size,
                                   anchor: anchor,
                                   direction: direction)
}

func placeShip(
    who: Who,
    size: Int,
    anchorRow: Int,
    anchorCol: Int,
    direction: Ship.Direction
) {

    let anchor = (row: anchorRow, col: anchorCol)
    let boardWho = getWhoBoard(who: who)
    boardWho.placeShip(size: size,
                        anchor: anchor,
                        direction: direction)
}
```

Add new type method to **Test** class

Add this code to test what you have just implemented:

```
class func testPlaceShip(engine: Engine) {
    func placeShipIfPossible(who: Engine.Who,
                              size: Int,
                              anchorRow: Int,
                              anchorCol: Int,
                              direction: Ship.Direction
    ) -> Bool {
```

```

    let x = engine.mayPlaceShip(who: who,
                                size: size,
                                anchorRow: anchorRow,
                                anchorCol: anchorCol,
                                direction: direction)

    if x {
        engine.placeShip(who: who,
                          size: size,
                          anchorRow: anchorRow,
                          anchorCol: anchorCol,
                          direction: direction)

        return true
    }

    return false
}

let player = Engine.Who.player
let opponent = Engine.Who.opponent

let shipsParams = [(who: player,
                     size: 4,
                     row: 3, col: 4,
                     direction: Ship.Direction.down),
                    (who: player,
                     size: 4,
                     row: 5, col: 6,
                     direction: Ship.Direction.left),
                    (who: opponent,
                     size: 4,
                     row: 3, col: 2,
                     direction: Ship.Direction.left),
                    (who: opponent,
                     size: 4,
                     row: 5, col: 6,
                     direction: Ship.Direction.up)]

for param in shipsParams {
    let result = placeShipIfPossible(
        who: param.who,
        size: param.size,
        anchorRow: param.row,
        anchorCol: param.col,
        direction: param.direction)

    print(result ? "OK" : "ERROR")
}
}

```

This method simply calls the `mayPlaceShip(size: anchor:direction:)` method from `Board` class. If there may be placed a ship of the size `size` starting at row `row` and column `col`, directed to `direction` direction then you physically put it on the board with the method `placeShip(size: anchor:direction:)` from `Board` class. Having this method, you may implement a loop to make sequence of tests (four in this case).

Modify `main.swift` file to do test

Replace code existing in `main.swift` file with the code below:

```
let game = Engine(rows: 10, cols: 10)
game.printBoards()
//game.test()
Test.testPlaceShip(engine: game)
game.printBoards()
```

After so many changes you may test if your code works as it is expected:

PLAYER

```
      1
    1234567890
  ++++++
1+.....+
2+.....+
3+.....+
4+.....+
5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
  ++++++
```

OPPONENT

```
      1
    1234567890
  ++++++
1+.....+
2+.....+
3+.....+
4+.....+
```

```

5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
+++++++

```

OK

ERROR

ERROR

OK

PLAYER

```

      1
    1234567890
+++++++
1+.....+
2+.....+
3+...X.....+
4+...X.....+
5+...X.....+
6+...X.....+
7+.....+
8+.....+
9+.....+
10+.....+
+++++++

```

OPPONENT

```

      1
    1234567890
+++++++
1+.....+
2+....X.....+
3+....X.....+
4+....X.....+
5+....X.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
+++++++

```

Program ended with exit code: 0

Code summary

In this chapter you made a lot of changes in different files. In reward you can now populate game board with ships.

Code completion

You will do:

In this part you will try to verify your knowledge and understanding of Swift you acquire so far. I will formulate a tasks for you, and you can try to implement them and next compare your solution with mine.

Tasks to complete, part 1

Now you have a knowledge which is sufficient to try your hand at implementing some functionalities. In this section I will formulate a tasks for you. In the next section you will find code you may use to test your solutions while in the third section you can find my solutions.

Task 1

According to assumptions from [Chapter 1: Initial steps,Section 1: Battleship](#) game, no ship can "touch" other ship. Your task is to modify `placeShip(size:anchor:direction:)` function from `Board` class in such a way that all cells directly surrounding ship are of `Board.CellType.notAllowed` type to prevent other ships to be placed too close (see figure below).

NOW	SHOULD BE
<pre> 1 1234567890 ++++++ 1+.....+ 2+.....+ 3+.....+ 4+...X.....+ 5+...X.....+ 6+.....+ 7+.....+ 8+.....+ 9+.....+ 10+.....+ ++++++</pre>	<pre> 1 1234567890 ++++++ 1+.....+ 2+.....+ 3+...+++.....+ 4+...+X+.....+ 5+...+X+.....+ 6+...+++.....+ 7+.....+ 8+.....+ 9+.....+ 10+.....+ ++++++</pre>

Task 2

When ship is totally destroyed (sunken) all its surrounding cells should be marked as it is showed at the figure below:

1		1		1
1234567890		1234567890		1234567890
+++++		+++++		+++++
1+.....+		1+.....+		1+.....+
2+.....+		2+.....+		2+.....+
3+.....+		3+.....+		3+..000.....+
4+...X.....+		4+...!.....+		4+..0!0.....+
5+...X.....+	==>	5+...X.....+	==>	5+..0!0.....+
6+.....+	hit at	6+.....+	hit at	6+..000.....+
7+.....+	(4, 4)	7+.....+	(5, 4)	7+.....+
8+.....+		8+.....+		8+.....+
9+.....+		9+.....+		9+.....+
10+.....+		10+.....+		10+.....+
+++++		+++++		+++++

This operation is safe because of assumption that no ship can "touch" other ship. Your task is to add to `Board` class method implementing this. Signature of this method is given below:

```
private func markWhenShipDestroyed(ship: Ship) {  
    // Mark all cells surrounding ship as ".rescue".  
}
```

Surrounding cells should be of `Board.CellType.rescue` type. Because this is a helper method called only from the body of its class, you prefix it with a `private` keyword.

Task 3

Implement a set of "shooting" methods:

1. In `Board` class implement `mayShot(row:col:) -> Bool` method to check if shot at coordinates (`row`, `col`) is possible.
2. In `Board` class implement `shot(row:col:)` method to make a shot at coordinates (`row`, `col`). For successful shot, call `afterHitAction(row:col:)` method (see next point).
3. In `Board` class implement `afterHitAction(row:col:)` private method. This method, if ship is destroyed, should call `markWhenShipDestroyed(ship:)` method.

Task 4

In `Board` class implement a method to automatically position all ships of a defined sizes. Signature of this method is given below:

```
func shipsAutoSetup(
    shipsSize: [Int],
    maxTriesPerShip: Int
) -> Int {
    // Try to position all ships
}
```

For every number defined in `shipSize` array you try at most `maxTriesPerShip` times to place ship of that size on a board. For every try you randomly select its anchor's row and column as well as direction. If it is possible to put the ship (this is checked with `mayPlaceShip(size:anchor:direction:)` method call) you place it (with `placeShip(size:anchor:direction:)` method call). As a result you return the number of ships which were successfully positioned on the board. If this number is equal to `shipSize` array it means that all ships were positioned successfully. To increase the chances of success in calling this method it is important to put bigger ships as first in `shipSize` array and smaller at the end.

To be able to position ships you may need a method returning an integer from given range (including both range ends) and such that is not contained in a set of excluded values to prevent from selecting occupied cells. This method is implemented for you below. It returns `nil` in case of failure or integer if it has been found. Add this method to `Utils` class.

```
class func getRandomInt(
  from: Int,
  to: Int,
  excluding: Set<Int>? = nil
) -> Int? {
  let maxTries = 10
  var candidate = -1

  guard from <= to else {return nil}

  if from == to {
    return from
  }

  for _ in 0 ..< maxTries {
    candidate = Int.random(in: from ... to)

    if let ex = excluding {
      if ex.contains(candidate) {
        return candidate
      }
    } else {
      return candidate
    }
  }

  return nil
}
```


Tests for part 1

When you complete all your tasks you should somehow verify them. Add to `Test` class the following code:

```
class func testShootingMethods(engine: Engine) {
    let player = Engine.Who.player

    var r = engine.mayPlaceShip(who: player,
                                size: 2,
                                anchorRow: 2,
                                anchorCol: 4,
                                direction: .down)

    if r {
        engine.placeShip(who: player,
                        size: 2,
                        anchorRow: 2,
                        anchorCol: 4,
                        direction: .down)

        r = engine.boardPlayer.mayShot(row: 3, col: 4)

        if r {
            engine.boardPlayer.shot(row: 3, col: 4)
        } else {
            print("Shot is not possible")
        }

        engine.printBoards()

        r = engine.boardPlayer.mayShot(row: 2, col: 4)

        if r {
            engine.boardPlayer.shot(row: 2, col: 4)
        } else {
            print("Shot is not possible")
        }
    }
}
```

```

        engine.printBoards()
    } else {
        print("Ship may not be placed")
    }
}

```

Then modify `main.swift` file to have the contents:

```

import Foundation

let game = Engine(rows: 10, cols: 10)
Test.testShootingMethods(engine: game)

```

If you run your code, you should see in console:

```

PLAYER
      1
    1234567890
  ++++++
1+.....+
2+...X.....+
3+...!.....+
4+.....+
5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
  ++++++
OPPONENT
      1
    1234567890
  ++++++
1+.....+
2+.....+
3+.....+
4+.....+
5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
  ++++++
PLAYER
      1

```

```

1234567890
+++++
1+..000.....+
2+..0!0.....+
3+..0!0.....+
4+..000.....+
5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
+++++
OPPONENT

```

```

1
1234567890
+++++
1+.....+
2+.....+
3+.....+
4+.....+
5+.....+
6+.....+
7+.....+
8+.....+
9+.....+
10+.....+
+++++

```

Program ended with exit code: 0

Next add to **Test** class another one type method:

```

class func testShipsAutoSetup(engine: Engine) {
    - = engine.boardPlayer.shipsAutoSetup(
        shipsSize: [4,3,3,2,2,2,1,1,1,1],
        maxTriesPerShip: 10
    )
    engine.printBoards()
}

```

In this method you try to put 10 ships: one of size 4, two of size 3, three of size 2 and four of size 1. All of them in a random locations and in a random direction.

Modify `main.swift` file to have the contents:

```

import Foundation

```

```
let game = Engine(rows: 10, cols: 10)
Test.testShipsAutoSetup(engine: game)
```

If you run your code, you should see result similar to the following (auto setup is a random process, so result after every run will be different):

PLAYER

```

1
1234567890
+++++
1++X+. . . . .+++
2++++. . . . .+X+
3+X+. . . . .+++X+
4+++ . . . . .+X+X+
5+X+. . +XX+X+++
6+++ . . . . .+X+
7++++ . . . . .+
8++X+. . +XXXX++
9++X+. . . . .+
10++X+. . +X+. . +
+++++
```

OPPONENT

```

1
1234567890
+++++
1+. . . . .+.
2+. . . . .+.
3+. . . . .+.
4+. . . . .+.
5+. . . . .+.
6+. . . . .+.
7+. . . . .+.
8+. . . . .+.
9+. . . . .+.
10+. . . . .+.
+++++
```

Program ended with exit code: 0

Tasks solutions, part 1

Task 1

```
func placeShip(
    size: Int,
    anchor: (row: Int, col: Int),
    direction: Ship.Direction
) {
    var modifier: (forRow: Int, forCol: Int)!
    var r: Int!
    var c: Int!
    var position = [(row: Int, col: Int, status: Ship.Status)]()

    switch direction {
    case .up:
        modifier = (forRow: -1, forCol: 0)
    case .down:
        modifier = (forRow: +1, forCol: 0)
    case .left:
        modifier = (forRow: 0, forCol: -1)
    case .right:
        modifier = (forRow: 0, forCol: +1)
    }

    for i in 0...size-1 {
        r = anchor.row + i*modifier.forRow
        c = anchor.col + i*modifier.forCol

        board[r][c] = CellType.ship
        position.append((row: r,
                        col: c,
                        status: .ready))

        // BEGIN: To add border along ship
        r = anchor.row+(modifier.forCol) + i*modifier.forRow
        c = anchor.col+(modifier.forRow) + i*modifier.forCol
        board[r][c] = .notAllowed
    }
}
```

```

    r = anchor.row-(modifier.forCol) + i*modifier.forRow
    c = anchor.col-(modifier.forRow) + i*modifier.forCol
    board[r][c] = .notAllowed
    // END: To add border along ship
}

// BEGIN: To add borders at the ends of ship
// Next to anchor
r = anchor.row + (-1)*modifier.forRow
c = anchor.col + (-1)*modifier.forCol
board[r][c] = .notAllowed

r = anchor.row+(modifier.forCol) + (-1)*modifier.forRow
c = anchor.col+(modifier.forRow) + (-1)*modifier.forCol
board[r][c] = .notAllowed

r = anchor.row-(modifier.forCol) + (-1)*modifier.forRow
c = anchor.col-(modifier.forRow) + (-1)*modifier.forCol
board[r][c] = .notAllowed

// Next to anchor oposit end
r = anchor.row + (size)*modifier.forRow
c = anchor.col + (size)*modifier.forCol
board[r][c] = .notAllowed

r = anchor.row+(modifier.forCol) + (size)*modifier.forRow
c = anchor.col+(modifier.forRow) + (size)*modifier.forCol
board[r][c] = .notAllowed

r = anchor.row-(modifier.forCol) + (size)*modifier.forRow
c = anchor.col-(modifier.forRow) + (size)*modifier.forCol
board[r][c] = .notAllowed
// END: To add borders at the ends of ship

ships.ships["\\(anchor)"] = Ship(
    size: size,
    position: position)
}

```

Task 2

```

private func markWhenShipDestroyed(ship: Ship) {
    let allCellsArround = [
        (rowModifier: -1, colModifier: 0), // top
        (rowModifier: -1, colModifier: +1), // top-right
        (rowModifier: 0, colModifier: +1), // right
        (rowModifier: +1, colModifier: +1), // bottom-right
        (rowModifier: +1, colModifier: 0), // bottom
        (rowModifier: +1, colModifier: -1), // bottom-left
        (rowModifier: 0, colModifier: -1), // left
    ]
}

```



```

    (rowModifier: -1, colModifier: -1)// top-left
]

var row, col: Int

for (r, c, _) in ship.position {
    for modifier in allCellsArround {
        row = r + modifier.rowModifier
        col = c + modifier.colModifier

        if board[row][col] == Board.CellType.empty ||
            board[row][col] == Board.CellType.shot ||
            board[row][col] == Board.CellType.notAllowed {
            board[row][col] = Board.CellType.rescue
        }
    }
}
}

```

Task 3

```

func mayShot(row: Int, col: Int) -> Bool {
    let yes: Set<Board.CellType> = [.empty,
                                    .ship,
                                    .notAllowed]

    if yes.contains(board[row][col]) {
        return true
    }

    let no: Set<Board.CellType> = [.hit,
                                   .rescue,
                                   .shot]

    if no.contains(board[row][col]) {
        return false
    }

    return false
}

func shot(row: Int, col: Int) {
    if board[row][col] == .empty ||
        board[row][col] == .notAllowed {
        board[row][col] = .shot
    } else if board[row][col] == .ship {
        board[row][col] = .hit
        afterHitAction(row: row, col: col)
    }
}

```

```

private func afterHitAction(row: Int, col: Int) {
    for (_, ship) in ships.ships {
        if ship.isLocatedAt(row: row, col: col) {
            ship.hitAt(row: row, col: col)
            if ship.isDestroyed {
                markWhenShipDestroyed(ship: ship)
            }
        }
    }
}

```

Task 4

```

func shipsAutoSetup(
    shipsSize: [Int],
    maxTriesPerShip: Int
) -> Int {
    var shipDirection = Ship.Direction.up
    var possible = true
    var success: Bool
    var positioned = 0
    var anchor: (row: Int, col: Int)

    for size in shipsSize {
        success = false
        for _ in 1...maxTriesPerShip {
            if let r = Utils.getRandomInt(from: 1, to: rows),
                let c = Utils.getRandomInt(from: 1, to: cols) {
                anchor = (row: r, col: c)

                if let direction = Utils.getRandomInt(from: 1, to: 4) {
                    switch direction {
                        case 1: shipDirection = .up
                        case 2: shipDirection = .right
                        case 3: shipDirection = .down
                        default: shipDirection = .left
                    }
                }
                possible = mayPlaceShip(
                    size: size,
                    anchor: anchor,
                    direction: shipDirection)

                if possible {
                    placeShip(
                        size: size,
                        anchor: anchor,
                        direction: shipDirection)
                    positioned += 1
                }
            }
        }
    }
}

```

```
        success = true
        break
    }
}
}
    }
    if !success {
        return positioned
    }
}
return positioned
}
```


Tasks to complete, part 2

In this section you will implement methods checking win conditions. After implementing whole game logic, finally you will be able to play the game.

Engine class modification

Add to **Engine** class three private properties to keep information about board dimension and sizes of ships used during the game (put them at the top of the file, just after enumerations):

```
private var shipsSize: [Int]
private var rows, cols: Int
```

and modify accordingly initializer:

```
init(rows: Int = 10, cols: Int = 10, shipsSize: [Int]) {
    self.boardPlayer = Board(rows: rows, cols: cols)
    self.boardOpponent = Board(rows: rows, cols: cols)

    self.shipsSize = shipsSize

    self.rows = rows
    self.cols = cols
}
```

Next add `getWhoTarget(who:)` method:

```
private func getWhoTarget(who: Who) -> Who {
    if who == Who.player {
```

```

    return Who.opponent
  }
  return Who.player
}

```

This method returns an opposite player.

Because previously implemented method `mayShot(row:col:)` is defined in `Board` class, you need a boilerplate method to call it:

```

func mayShot(who: Who, row: Int, col: Int) -> Bool {
  let boardWho = getWhoBoard(who: getWhoTarget(who: who))
  return boardWho.mayShot(row: row, col: col)
}

```

If you know that shot is possible, you can shoot:

```

func shot(who: Who, row: Int, col: Int) {
  let boardWho = getWhoBoard(who: getWhoTarget(who: who))
  boardWho.shot(row: row, col: col)
}

```

Last boilerplate method left to implemented calls ship auto-layout metod from `Board` class:

```

func shipsAutoSetup(
  shipsSize: [Int],
  maxTriesPerShip: Int,
  who: Who
) -> Bool {
  let boardWho = getWhoBoard(who: who)
  let number = boardWho.shipsAutoSetup(
    shipsSize: shipsSize,
    maxTriesPerShip: maxTriesPerShip
  )

  if shipsSize.count == number {
    return true
  }

  return false
}

```

Task 5

To make playing game possible you need a method used to check if there is a winner and, if the answer is *yes*, who is this. This method should have a following signature:

```
func checkWhoWins() -> Who? {  
}
```

Add this method to **Engine** class.

Task 6

Implement a very simple shooting method. In this case you try at most **maxTries** times to find randomly an acceptable cell (that is a cell you are allowed to shoot on). If you fail, you systematically, row by row and column by column, search until you find acceptable cell. Signature of this method is given below:

```
func getShotCoordinatesForOpponent(  
    maxTries: Int  
) -> (row: Int, col: Int)? {  
}
```

To get random values you may use

getRandomInt(from:to:excluding:) method from **Utils** class.
Add this method to **Engine** class.

Task 7

Delete all the contents from `main.swift` file leaving only **import Foundation**. Next add to this file function with the following signature:

```
func getIntFromCommandLine(
    message: String,
    rangeMin: Int,
    rangeMax: Int
) -> Int {
}

```

This function should print a **message** and then ask a (human) player to provide integer laying in closed interval `[rangeMin, rangeMax]`. It should keep asking as long as number entered by a player is outside a given range.

Task 8

Based on function from **Task 7** define in `main.swift` file a function getting shot coordinates from human player. This function should have a following signature:

```
func getShotCoordinatesForPlayer(
    engine: Engine,
    maxRows: Int,
    maxCols: Int
) -> (row: Int, col: Int)? {
}

```

`main.swift` file modification

To `main.swift` file, just after methods from **Task 7** and **Task 8** add a set of variables:

- Array with sizes of the ships

```
let shipsSize = [4, 3, 3, 2, 2, 2, 1, 1, 1, 1]

```


- An instance of game engine:

```
let game = Engine(shipsSize: shipsSize)
```

- Weather ships auto-layout for an opponent (computer) succeeded:

```
let opponentReady = game.shipsAutoSetup(  
    shipsSize: shipsSize,  
    maxTriesPerShip: 20,  
    who: .opponent  
)
```

- Weather ships auto-layout for a player (human) succeeded:

```
let playerReady = game.shipsAutoSetup(  
    shipsSize: shipsSize,  
    maxTriesPerShip: 20,  
    who: .player  
)
```

- Indicating whose turn is currently:

```
var whoseTurn = Engine.Who.player
```

- Player's or opponent's shot coordinates:

```
var coordinates: (row: Int, col: Int)?
```

Now you are ready to implement final part of code – a main game loop:

```
if opponentReady, playerReady {  
    game.printBoards()  
    while(true){  
        print("\n\n\n TURN")  
        print(whoseTurn)  
  
        if whoseTurn == Engine.Who.player {  
            coordinates = getShotCoordinatesForPlayer(  
                engine: game,  
                maxRows: 10,  
                maxCols: 10  
            )  
        }  
    }  
}
```

```

        if coordinates == nil {
            print("Player can't shoot")
            break
        } else {
            print("Shot at row \(coordinates!.row) and column \(
(coordinates!.col)")
        }

        game.shot(
            who: whoseTurn,
            row: coordinates!.row,
            col: coordinates!.col
        )
        whoseTurn = Engine.Who.opponent
    } else {
        coordinates = game.getShotCoordinatesForOpponent(
            maxTries: 100
        )
        if coordinates == nil {
            print("Opponent can't shoot")
            break
        } else {
            print("Shot at row \(coordinates!.row) and column \(
(coordinates!.col)")
        }

        game.shot(
            who: whoseTurn,
            row: coordinates!.row,
            col: coordinates!.col
        )
        whoseTurn = Engine.Who.player
    }

    game.printBoards()
    if let who = game.checkWhoWins() {
        print("The winner is \(who)")
        break
    }
} else {
    print("Can't position all ships")
}

```

Now you can run your code. Ships for player (human player) and his opponent (computer player) should be automatically positioned on game boards. Both players should have 10 ships. To make debug possible, both boards are printed on the screen. Happy playing!

Tasks solutions, part 2

Task 5

```
func checkWhoWins() -> Who? {
    if boardPlayer.ships.shipsAtCommand == 0 {
        return Who.opponent
    } else if boardOpponent.ships.shipsAtCommand == 0 {
        return Who.player
    }

    return nil
}
```

Task 6

```
func getShotCoordinatesForOpponent(
    maxTries: Int
) -> (row: Int, col: Int)? {
    var row, col: Int?

    // Use random approach
    for _ in 1...maxTries {
        row = Utils.getRandomInt(from: 1, to: rows)
        col = Utils.getRandomInt(from: 1, to: cols)

        if let r = row, let c = col {
            if mayShot(who: Who.opponent, row: r, col: c) {
                return (row: r, col: c)
            }
        }
    }

    // If previous failed, use systematic search approach
    for r in 0...rows+1 {
        for c in 0...cols+1 {
```

```

        if mayShot(who: Who.opponent, row: r, col: c) {
            return (row: r, col: c)
        }
    }
}

return nil
}

```

Task 7

```

func getIntFromCommandLine(
    message: String,
    rangeMin: Int,
    rangeMax: Int
) -> Int {
    print(message)
    while(true) {
        if let input = readLine() {
            if let int = Int(input) {
                if int >= rangeMin && int <= rangeMax {
                    return int
                } else {
                    ("\"(input) is not in range [\"(rangeMin)-\"(rangeMax)].
Please try again")
                }
            } else {
                print("\"(input) is not a valid integer. Please try
again")
            }
        }
    }
}

```

Task 8

```

func getShotCoordinatesForPlayer(
    engine: Engine,
    maxRows: Int,
    maxCols: Int
) -> (row: Int, col: Int)? {
    while(true) {
        let row = getIntFromCommandLine(
            message: "Enter row",
            rangeMin: 1,
            rangeMax: maxRows)
    }
}

```

```
let col = getIntFromCommandLine(  
  message: "Enter column",  
  rangeMin: 1,  
  rangeMax: maxCols)  
  
if engine.mayShot(who: .player, row: row, col: col) {  
  return (row: row, col: col)  
}  
}  
}
```


Structures, inheritance and errors handling

You will do:

In this part you will extend your knowledge with information on more advanced topics.

You will learn:

- What a **structure** is and how it differs from **class**.
- **Inheritance** with **type checking** and **access control**.
- How to handle **exceptional** situation.

Structures

Structures are very similar to *classes*. Both are general-purpose, flexible constructs that become the building blocks of your program's code. As you have seen so far, you define properties and methods to add functionality to your classes using the same syntax you use to define constants, variables, and functions; the same you do in case of structures. With **structures and classes** in Swift you can

- define *properties* to store values;
- define *functions* to provide functionality;
- define *initializers* to set up their initial state;
- be *extended* to expand their functionality beyond a default implementation;
- conform to *protocols* to provide standard functionality of a certain kind;
- define *subscripts* to provide access to their values using subscript syntax.

Moreover, **classes have some additional capabilities** that structures do not:

- inheritance enabling one class to inherit the characteristics of another;

- you can check and interpret the type of a class instance at runtime;
- reference counting allows more than one reference to a class instance;
- deinitializers enable an instance of a class to free up any resources it has assigned.

Another worth mention difference is that **structures are always copied** when they are passed around in the code and to not use reference counting. **Structure instances are always passed by value, and class instances are always passed by reference.**

The additional capabilities that classes support come at the cost of increased complexity. As a general guideline, prefer structures because they're easier to reason about, and use classes when they're appropriate or necessary. In practice, this means most of the custom data types you define will be structures and enumerations.

As you know, for classes you cannot leave properties uninitialized – you have to:

- set their values explicitly,
- or define them as optional,
- or provide initializer to set their values.

```
class ClassOption1 {  
    var property = 0  
}  
  
class ClassOption2 {  
    var property: Int?  
}  
  
class ClassOption3 {  
    var property: Int  
  
    init(property: Int) {  
        self.property = property  
    }  
}
```

```

    }
}

// Cause an error:
// Class 'ClassOption4' has no initializers
class ClassOption4 {
    var property: Int
}

var ico1 = ClassOption1()
var ico2 = ClassOption2()
var ico3 = ClassOption3(property: 13)

```

The same rules are applied to structures but in this case you have also another one option: *automatically generated memberwise initializers*.

```

struct StructOption1 {
    var property = 0
}

struct StructOption2 {
    var property: Int?
}

struct StructOption3 {
    var property: Int

    init(property: Int) {
        self.property = property
    }
}

// Cause NO error:
struct StructOption4 {
    var property: Int
}

var iso1 = StructOption1()
var iso2 = StructOption2()
var iso3 = StructOption3(property: 23)
var iso4 = StructOption4(property: 24)

```

You can use automatically generated memberwise initializer in case of implicitly initialized structure:

```
iso1 = StructOption1(property: 21)
```

Of course, if initializer is required, you may not skip it:

```
// Missing argument for parameter 'property' in call
var iso5 = StructOption4()
```

If class or struct is declared with **var** you can modify its properties:

```
ico1.property = 11
iso1.property = 21
```

If class or struct is declared with **let** situation is a little bit different: you are not allowed to modify its properties:

```
let c = ClassOption1()
let s = StructOption1()

c.property = 11

// Cause an error:
// Cannot assign to property: 's' is a 'let' constant
s.property = 21
```

Mutating methods

For structures (and also enumeration) you can implement *mutating methods* allowing to modify the properties of value type (structures and enumerations but not classes which are reference type) from within its instance methods. **By default, the properties of a value type cannot be modified from within its instance methods.** You have to use a special **mutating** keyword to implement such a method:

```
struct Item {
    var item: String

    func nonMutate() -> String {
        return item
    }

    mutating func mutate(_ value: String) {
        item = value
    }
}
```

```

    // Invalid mutating function
    //func mutateInvalid(_ value: String) {
    //    item = value
    //}
}

var i = Item(item: "Test string")
print("Item \(i.item)")
// Prints:
// Item Test string

i.item = "Another text"
print(i.nonMutate())
// Prints:
// Another text

print(i.item)
// Prints:
// Another text

i.mutate("Again new text")
print(i.nonMutate())
// Prints:
// Again new text

print(i.item)
// Prints:
// Again new text

```

If you try to skip **mutating** modifier in **mutate(_:)** method, you will get an error:

Cannot assign to property: 'self' is immutable

So you can modify properties from outside of structure with statement like that:

```
i.item = "Another text"
```

but you are not allowed to do this via function call:

```
i.mutateInvalid("Try it")
```

if this function is not marked with `mutating` keyword.

Inheritance

I hope that you have an understanding what an inheritance is, so I'll give only basic syntax information. In Swift subclass definition is of the form:

```
class Subclass: Superclass {  
    // Subclass definition goes here  
}
```

Have in mind that multiple inheritance is not allowed in Swift:

```
class A {  
    var a = 1  
}  
  
class B {  
    var b = 2  
}  
  
class C: A {  
    var c = 3  
}  
  
// Cause an error:  
// Multiple inheritance from classes 'A' and 'B'  
class D: A, B {  
    var d = 4  
}  
  
var a = A()  
var c = C()  
  
a.a = 5  
  
// Proof that inside class C there are all "things"  
// from class A
```

```
c.c = 7
c.a = 9
```

You use **override** keyword to mark method, property or subscript as *overriden*. Superclass's method, property or subscript is accessed with **super** keyword. To prevent method, property or subscript from being overridden you mark it with **final** modifier. You can mark entire class as final preventing them to be subclassing.

```
class BaseClass {
    var somePropertyInt: Int
    init () {
        somePropertyInt = 5
        print("BC: init complete");
    }

    func doSomething(){
        print("BC: doSomething")
    }

    final func notForSubclassing(){
        print("BC: notForSubclassing")
    }

    func anotherOneFunction(){
        print("BC: anotherOneFunction")
    }
}

class SubClass: baseClass {
    var somePropertyString: String
    override init () {
        somePropertyString = "text"
        print("SC: init complete");
    }

    override func doSomething(){
        print("SC: doSomething")
    }

    func callMethodFromSuperClass() {
        print("SC: callMethodFromSuperClass")
        super.notForSubclassing()
    }
}

var sc = SubClass()
// Prints:
```

```

// SC: init complete
// BC: init complete

sc.doSomething()
// Prints:
// SC: doSomething

sc.notForSubclassing()
// Prints:
// BC: notForSubclassing

sc.anotherOneFunction()
// Prints:
// BC: anotherOneFunction

print("\({sc.somePropertyInt}) \({sc.somePropertyString}")
// Prints:
// 5 text

sc.callMethodFromSuperClass()
// Prints:
// SC: callMethodFromSuperClass
// BC: notForSubclassing

```

In the above example, if you change **SubClass**'s initializer to the form:

```

override init () {
    somePropertyString = "text"
    super.init()
    print("subClass: init complete");
}

```

then creating **sc** variable will result with:

```

var sc = subClass()
// Prints:
// baseClass: init complete
// subClass: init complete

```

Note that before you call in initializer **self.init()** you should initialize all properties. If you try to type:

```

override init () {
    super.init()
    somePropertyString = "text"
}

```

```
    print("subClass: init complete");  
}
```

you will get an error:

**Property 'self.somePropertyString' not initialized at
super.init call**

As you know from previous section, inheritance only applies to classes. However it is possible to get inheritance-like behavior also for structures if you use protocols – see [Chapter 9: Protocols and generics, Section 2: Structs and inheritance](#).

Type checking and casting

In Swift we have two special types for working with indefinite, or better say: *any* type:

- **Any** which can represent an instance of any type at all (including function types);
- **AnyObject** which can represent an instance of any class type.

The most basic example of **Any** usage is an array to store items of any type:

```
var arrayOfAnyInstances = [Any]()

arrayOfAnyInstances.append(5)
arrayOfAnyInstances.append(1.23)
arrayOfAnyInstances.append("test")
arrayOfAnyInstances.append(
    {(arg: String) -> String in
        "Echo: \(arg)"
    })

print(arrayOfAnyInstances)
// Prints: [5, 1.23, "test", (Function)]
```

The **Any** type represents values of *any* type, including optional types. Swift gives you a warning if you use an optional value where a value of type **Any** is expected. If you really do need to use an optional value as an **Any** value, you can use the **as** operator to explicitly cast the optional to **Any**:

```

let optionalNumber: Int? = 5
arrayOfAnyInstances.append(optionalNumber)
// Warning:
// Expression implicitly coerced from 'Int?' to 'Any'
arrayOfAnyInstances.append(optionalNumber as Any)

```

To check whether an instance is of a certain subclass type, use the type check operator: **is**. This operator returns true if the instance is of that subclass type and false if it is not.

```

for item in arrayOfAnyInstances {
    if item is Int {
        print("\(item) is of Int type");
    } else if item is String {
        print("\(item) is of String type");
    } else {
        print("\(item) is of \(type(of: item)) type");
    }
}
// Prints:
// 5 is of Int type
// 1.23 is of Double type
// test is of String type
// (Function) is of (String) -> String type
// Optional(5) is of Int type
// Optional(5) is of Int type

```

A constant or variable of a certain class type may actually refer to an instance of a subclass. Other words, **a constant or variable of a certain class type may point to any class down the hierarchy; pointer dedicated to use for certain class may point to any child of this class (to any subclass):**

```

class A {}

class B: A {}

class C: B {}

var iA = A()
var iB = B()
var iC = C()

var pointer: B

pointer = iB

```

```
// Error:
// // Cannot assign value of type 'A' to type 'B'
pointer = iA
pointer = iC
```

You can try to *downcast* to the subclass type with a type cast operator:

- in the conditional form **as?** when returns an optional value of the type you try to downcast to;
- in the forced form **as!** to attempt the downcast and force unwraps the result as a single compound action.

Conditional cast to the superclass (*upcast*) always succeeds.

With pointers and casting you can treat an object being and instance of a given class as an object of different class which is its ancestor. In programming languages the existence of a single interface to entities of different types or the use of a single symbol to represent multiple different types is called *polymorphism*. This is one of the fundamental building block of every object-oriented programming language. With polymorphism your code can be agnostic as to which class in the supported hierarchy (*family* of objects) it is operating on – the parent class or one of its descendants.

See examples below (the name **Test** preceding class name in **Test.B** and similar statements is the name of my project where I put this code): if you think the first is too synthetic don't hesitate and jump to the second which I hope is much more clear.

Example 1:

```
// Part 1
var b = B()
print("1: \(b) is of \(type(of: b)) type");
// Prints:
// 1: Test.B is of B type

// Part 2.1
```

```

var x = B() as? A
// Warning:
// Conditional cast from 'B' to 'A' always succeeds

print("2: \((x ?? A()) is of \((type(of: x)) type");
// Prints:
// 2: test.B is of Optional<A> type
// Default value is provided to avoid warning:
// String interpolation produces a debug description for an
// optional value; did you mean to make this explicit?

if let c = x {
    print("3: \((c) is of \((type(of: c)) type");
} else {
    print("3: Casting problems")
}
// Prints:
// 3: Test.B is of B type

// Part 2.2
// Because conditional cast to the superclass always succeeds,
// you can use 'as'
var xForced = B() as A
print("4: \((xForced) is of \((type(of: xForced)) type");
// Prints:
// 4: Test.B is of B type

// Warning:
// 'is' test is always true
if xForced is A {
    print("5: Is A class")
}

if xForced is B {
    print("6: Is B class")
}

if xForced is C {
    print("7: Is C class")
}
// Prints:
// 5: Is A class
// 6: Is B class

// Part 3
var y = B() as? C
print("8: \((y ?? C()) is of \((type(of: y)) type");
// Prints:
// 8: Test.C is of Optional<C> type

if let c = y {
    print("9: \((c) is of \((type(of: c)) type");
}

```

```

} else {
    print("9: Casting problems")
}
// Prints:
// 9: Casting problems

// Part 4
var pA: A
pA = iB
var t1 = pA as? B
if let c = t1 {
    print("10: \ (c) is of \ (type(of: c)) type");
} else {
    print("10: Casting problems")
}
// Prints:
// 10: Test.B is of B type

var t2 = pA as? C
if let c = t2 {
    print("11: \ (c) is of \ (type(of: c)) type");
} else {
    print("11: Casting problems")
}
// Prints:
// 11: Casting problems

```

Example 2:

Imagine that you have a secret keeping application you store your passwords, credit cards data, various access code, etc. Consider a following hierarchy of objects:

```

class Secret {
    var name: String

    init(name: String) {
        self.name = name
    }
}

class Email: Secret {
    var email: String
    var password: String

    init(name: String, email: String, password: String) {
        self.email = email
    }
}

```

```

        self.password = password
    }
    super.init(name: name)

    func printEmail() {
        print("email: \(email)\npassword: \(password)")
    }
}

class CreditCard: Secret {
    var number: String
    var cvvCode: String

    init(name: String, number: String, cvvCode: String) {
        self.number = number
        self.cvvCode = cvvCode

        super.init(name: name)
    }

    func printCreditCard() {
        print("number: \(number)\ncvvCode: \(cvvCode)")
    }
}

```

Having this you can create an array to keep some of your secrets and fill it with mock data:

```

var allMySecrets = [Secret]()

allMySecrets.append(Email(name: "Private email",
                           email: "private@domain.com",
                           password: "123abc"))
allMySecrets.append(CreditCard(name: "Bank 1",
                                number: "1234-5678",
                                cvvCode: "123"))
allMySecrets.append(Email(name: "Job email",
                           email: "my.job@email.server.com",
                           password: "123abc"))
allMySecrets.append(CreditCard(name: "Bank 2",
                                number: "9876-5432",
                                cvvCode: "321"))

```

Now you can create a pointer and set it to point to one of your secrets:

```

var currentSecret: Secret
currentSecret = allMySecrets[2]

```

Which secret is now pointed by `currentSecret`? It hard to guess. You have to check it. You may try to downcast to one of possible subclasses:

```
var t = currentSecret as? Email
```

If the result is not nil, then downcasting was successful and you may treat your object as an instance of a given type:

```
if let e = t {  
    print("This is an email")  
    e.printEmail()  
} else {  
    print("This is NOT an email")  
}  
// Prints:  
// This is an email  
// email: my.job@email.server.com  
// password: 123abc
```

If you want, you cant print them all:

```
print("=== All my secrets ===")  
  
for s in allMySecrets {  
    switch s {  
    case is Email:  
        let e = s as! Email  
        e.printEmail()  
    case is CreditCard:  
        let cc = s as! CreditCard  
        cc.printCreditCard()  
    default:  
        print("Unknown secret")  
    }  
}
```

In this case you will see:

```
=== All my secrets ===  
This is an email  
email: private@domain.com  
password: 123abc
```

This is a credit card
number: 1234-5678
cvcCode: 123
This is an email
email: my.job@email.server.com
password: 123abc
This is a credit card
number: 9876-5432
cvcCode: 321

Access control

Access control restricts access to/from parts of your code. With this feature you can hide the implementation details, and enable access to it with a preferred interface through which that code can be used. Swift provides five different access levels. These access levels are relative to the source file in which an entity is defined, and also relative to the module that source file belongs to.

- *Open access* (**open** keyword) and *public access* (**public**) enable entities to be used within any source file from their defining module, as well as in a source file from another module that imports the defining module. See below for difference between open and public access.
- *Internal access* (**internal**) enables entities to be used within any source file from their defining module, but not in any source file outside of that module. This is default access specifier in Swift.
- *File-private access* (**fileprivate**) restricts the use of an entity to its own defining source file.
- *Private access* (**private**) restricts the use of an entity to the enclosing declaration.

Open access differs from public access as follows:

- Open access applies only to classes and class members.

- Classes with public access, or any more restrictive access level, can be subclassed only within the module where they're defined.
- Class members with public access, or any more restrictive access level, can be overridden by subclasses only within the module where they're defined.
- Open classes can be subclassed within the module where they're defined, and within any module that imports the module where they're defined.
- Open class members can be overridden by subclasses within the module where they're defined, and within any module that imports the module where they're defined.

More important rules restricting access:

- **Rule 1** Almost all entities in your code, if we do not specify an explicit access level itself, have a default access level of *internal*.
- **Rule 2** The access control level of a type also affects the default access level of that type's members: properties, methods, initializers, and subscripts. For example, having defined type with a *private* access level, the default access level of its members will also be *private*.
- **Rule 3** A *public* type defaults for its members is *internal*. This ensures that the open to the public API for a type is something you agree to publishing, and avoids presenting the internal workings details of a type as public API by mistake.
- **Rule 4** A tuple type's access level is deduced automatically when the tuple type is used, and can't be specified explicitly.
- **Rule 5** The access level for a function type is calculated as the most restrictive access level of the function's parameter types and return type. You must specify the access level explicitly as part of the

function's definition if the function's calculated access level doesn't match the contextual default.

- **Rule 6** Nested types defined within a *private* (or *file-private*) type have an automatic access level of *private* (or *file-private*). Nested types defined within a *public* type or an *internal* type have an automatic access level of *internal*.
- **Rule 7** A subclass can't have a less restrictive access level than its superclass. For example, we can't write a *public* subclass of an *internal* superclass.
- **Rule 8** An override can make an inherited class member more accessible than its superclass version.
- **Rule 9** A constant, variable, or property can't be more public than its type. For example it's not valid to have a *public* property with a *private* type.

As a general rule in Swift we have that no entity can be defined in terms of another entity that has a less restrictive access level. For example, a public variable cannot be defined as having an internal private type, because that private type might not be available everywhere that the public variable.

```
// Compile Error:
// Only classes and overridable class members can be declared
// 'open'; use 'public'
open var variableOpen = 0
public var variablePublic = 0
// Implicitly internal
var variableInternal = 0
fileprivate var variableFilePrivate = 0
private var variablePrivate = 0

open class classOpen{}

// Explicitly public class
public class classPublic {
    // Explicitly public class member
    public var propertyPublic = 0
}
```

```

// By default internal (Rule 3)
var propertyInternal = 0
// Explicitly file private class member
fileprivate func methodFilePrivate() {}
// Explicitly private class member
private var propertyPrivate = 0
}

// By default internal (Rule 1)
class classInternal {
    // By default internal (Rule 2)
    var propertyInternal = 0
    // Explicitly file private class member
    fileprivate func methodFilePrivate() {}
    // Explicitly private class member
    private var propertyPrivate = 0
}

// Explicitly file private class
fileprivate class classFilePrivate {
    // By default file private (Rule 2)
    func methodFilePrivate() {}
    // Explicitly private class member
    private var propertyPrivate = 0
}

// Explicitly private class
private class classPrivate {
    // By default private (Rule 2)
    var variable = 0
    private var variableExplicitlyPrivate = 0

    // Rule 6 - this nested type has an automatic
    // access level of private; see below
    class classPrivateNestedType {}
    var xx = classPrivateNestedType()
}

// Rule 9
private var xx = classPrivate()
// Without private:
// Compile Error: Variable must be declared private
// or fileprivate because its type
// 'classPrivate.classPrivateNestedType' uses a private type
private var yy = xx.xx
// Compile Error: 'variableExplicitlyPrivate' is inaccessible
// due to 'private' protection level
//var ww = xx.variableExplicitlyPrivate

// ??? According to Rule 2 variable should be private
// but it is not. Why? ???

```

```

var zz = xx.variable

// Without private:
// Compile Error: Function must be declared private
// or fileprivate because its results uses a private type
// Rule 4 and 5
private func someFunction() -> (classPublic, classPrivate)
{ ... }

// Rule 7
// Compile Error: Class cannot be declared public
// because its superclass is private
public class classPublicWithPrivateSuperclass: classPrivate {
    fileprivate func someMethod() {}
}

// Rule 8
internal class classInternalOverrides: classPublic {
    override internal func methodFilePrivate() {
        super.methodFilePrivate()
    }
}

```


Errors handling

In Swift, **errors** are represented by values of types that conform to the **Error** protocol. This empty protocol indicates that a type can be used for error handling. A good choice to represent a group of related error types are enumerations. A **throw** keyword is used to bring to life an error.

```
enum ErrorCollection: Error {  
    case errorType0  
    case errorType1  
    case errorType2  
}  
  
throw ErrorCollection.errorType1
```

When an error is thrown, some piece of code must be responsible for handling it. There are four ways to handle errors in Swift.

- Error can be propagated from a function to the code that calls that function.
- Error can be handled with **do-catch** statement.
- Error can be handled as an optional value.
- Error propagation can be disabled and its call wrapped in a runtime assertion that no error will be thrown.

ERROR PROPAGATION

You use the **throws** keyword to indicate that a function, method, or initializer can throw an error. A function marked this way is called a *throwing function*. Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside that function.

```
class classThrowingErrors {
  func functionThrowingErrors(
    forNumber number: Int
  ) throws -> Int {
    if (number == 0) {
      throw ErrorCollection.errorType0
    }
    print("number: \(number)")
  }
}
```

ERROR HANDLING WITH do-catch

do-catch has the following general form:

```
do {
  try expression
  // statements
} catch pattern_1 {
  // statements
} catch pattern_2 where condition {
  // statements
}
```

Example below shows very basic example how it can be used:

```
class classThrowingErrors {
  func functionThrowingErrors(
    forNumber number: Int
  ) throws -> Int {
    if (number == 0) {
      throw ErrorCollection.errorType0
    }
    print("number: \(number)")
    return number
  }

  func functionNoThrowingErrors(forNumber number: Int) {
```

```

do {
  let x = try functionThrowingErrors(forNumber: 0)
  print("No errors, result is \(x)")
} catch ErrorCollection.errorType0 {
  print("Error type 0")
} catch ErrorCollection.errorType1 {
  print("Error type 1")
} catch ErrorCollection.errorType2 where number > 4 {
  print("Error type 2")
} catch let error {
  // Who knows, maybe there are more errors possible?
  // We have to catch all of them.
  print(error.localizedDescription)
}
}

var c = classThrowingErrors()
c.functionNoThrowingErrors(forNumber: 0)

```

This code prints the message:

Error type 0

ERROR HANDLING WITH OPTIONALS

You use `try?` to handle an error by converting it to an optional value. If an error is thrown while evaluating the `try?` expression, the value of the expression is `nil`.

```
var e = try? c.functionThrowingErrors(forNumber: 0)
```

DISABLE ERROR PROPAGATION

When you are sure that throwing function or method will not throw an error at runtime you can write `try!` to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. Note that if an error actually occurs and is thrown, you will get a runtime error and your application will be terminated immediately.

```
var t = try! c.functionThrowingErrors(forNumber: 1)
```

In Java very common statements sequence is `try-catch-finally`. The `finally` block *always* executes when the `try` block exits. This ensures that the `finally` block is executed even if an unexpected exception occurs. But `finally` is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a `return`, `continue`, or `break`. Putting cleanup code in a `finally` block is always a good practice, even when no exceptions are anticipated. [JAVADOC:1]

In Swift there is very similar statement: `defer`. It is used to execute a set of statements just before code execution leaves the current block of code. As in Java with this statement you can do any necessary cleanup that will be performed regardless of how execution leaves the current block of code – whether it leaves because an error was thrown or because of a statement such as `return` or `break` or even quite natural without any errors or jumps. The most basic example given in all tutorials is when you want to ensure that file descriptors are closed or manually allocated memory is freed.

A `defer` statement defers execution until the current scope is exited. The deferred statements may not contain any code that would transfer control out of the statements, such as a `break` or a `return` statement, or by throwing an error. Deferred actions are executed in the reverse of the order that they're written in your source code. That is, the code in the first `defer` statement executes last, the code in the second `defer` statement executes second to last, and so on. The last `defer` statement in source code order executes first.

Analyze carefully the following example:

```
func doSomethingWithResources() throws {  
    print("1")  
    defer {  
        print("3")  
    }  
}
```

```

defer {
    print("5")
}
print("2")
throw ErrorCollection.errorType0
// Warning:
// Code after 'throw' will never be executed
print("6")
// Warning:
// 'defer' statement at end of scope always executes
// immediately; replace with 'do' statement to silence
// this warning
defer {
    print("4")
}
}

defer {
    print("9")
}
do {
    try doSomethingWithResources()
} catch {
    print("7")
    throw ErrorCollection.errorType0
}
defer {
    print("8")
}
}

```

You will see the following output:

```

1
2
5
3
7
9
Fatal error: Error raised at top level:
[...]
```


Protocols and generics

You will do:

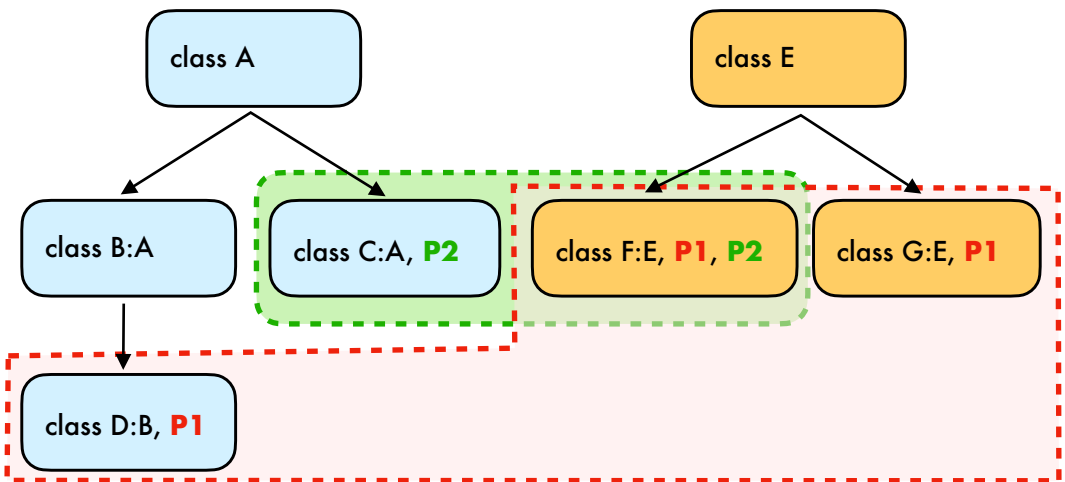
In this part you will continue extending your knowledge with information on more advanced topics.

You will learn:

- For what and how you can use **protocols**.
- How you can write general purpose code with **generics**.

Protocols

A *protocol* is a group of related properties and methods that can be implemented by any class. With protocols you can define a one single API and use it (after implementation) in *hierarchically unrelated* classes. This way you can represent *horizontal relationships* on top of an existing tree-like class hierarchy:



In this sense protocols are like Java *interfaces*. A syntax of protocol is shown below:

```
protocol ProtocolName {  
    // Protocol definition goes here  
}
```

Here is a more complicated syntax for class `ClassName` having a superclass `SuperclassName` conforming to protocol `ProtocolName1` as well as `ProtocolName2`:

```
class ClassName: SuperclassName, ProtocolName1, ProtocolName2 {  
    // Protocol definition goes here  
}
```

As you can see, every class may have at most one superclass but may implement many protocols.

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn't specify whether the property should be a stored property or a computed property. The protocol also specifies whether each property must be gettable or gettable and settable.

Protocols can require specific instance methods and type methods to be implemented. These methods are written as part of the protocol's definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body. Variadic parameters are allowed, subject to the same rules as for normal methods. Default values, however, can't be specified for method parameters within a protocol's definition.

It's sometimes necessary for a method to modify (or mutate) the instance it belongs to. In such a case, if you have a structure or enumerations, you place the `mutating` keyword before a method's `func` keyword to indicate that the method is allowed to modify the instance it belongs to and any properties of that instance. If you mark a protocol instance method requirement as `mutating`, you don't need to write the mutating keyword when writing an implementation of that method for a class.

Protocols can require specific initializers to be implemented. We write these initializers as part of the protocol's definition in exactly the same way as for normal initializers, but without curly braces or an initializer body. When implemented, a protocol initializer must be marked with the **required** keyword.

That was a lot of theory, so now see how it works in the following example:

```
protocol ProtocolForClass {
    var mustBeSettable: Int { get set }
    var maybeSettable: Int { get }

    func methodName() -> Int

    func iAmAllowedToModifyThisInstance(withValue value:Int)

    init(someParameter value: Int)
}

protocol ProtocolForStructure {
    var mustBeSettable: Int { get set }
    var maybeSettable: Int { get }

    func methodName() -> Int

    // We need mutating to work with structures and enumerations
    // We can use it also for classes
    mutating func iAmAllowedToModifyThisInstance(withValue
value:Int)

    init(someParameter value: Int)
}

class ClassImplementingProtocol: ProtocolForClass {
    // The following is perfectly legal
    var mustBeSettable: Int = 0

    // This is also correct
    // private var accumulator: Int = 0
    // var mustBeSettable: Int {
    //     get {
    //         accumulator
    //     }
    //     set {
    //         accumulator = newValue
    //     }
    // }
```

```

// }

// This is NOT correct
// Error
// Type 'ClassImplementingProtocol' does not conform
//to protocol 'ProtocolForClass'
// private var accumulator: Int = 0
// var mustBeSettable: Int {
//     get {
//         accumulator
//     }
// }
// }
var maybeSettable: Int = 0

func methodName() -> Int {
    return mustBeSettable + maybeSettable
}

func iAmAllowedToModifyThisInstance(withValue value: Int) {
    mustBeSettable = value
    maybeSettable = value * 2
}

// Error without required
// Initializer requirement 'init(someParameter:)' can only
// be satisfied by a 'required' initializer in non-final
// class 'ClassImplementingProtocol'
required init(someParameter value: Int) {
    mustBeSettable = value
    maybeSettable = value * 2
}
}

struct StructureImplementingProtocol: ProtocolForStructure {
    var mustBeSettable: Int = 0
    var maybeSettable: Int = 0

    func methodName() -> Int {
        return mustBeSettable + maybeSettable
    }

    mutating func iAmAllowedToModifyThisInstance(
        withValue value: Int
    ) {
        mustBeSettable = value
        maybeSettable = value * 2
    }

    init(someParameter value: Int) {
        mustBeSettable = value
        maybeSettable = value * 2
    }
}

```

```

}

var c = ClassImplementingProtocol(someParameter: 2)
print(c.methodName()) // Prints: 6
c.iAmAllowedToModifyThisInstance(withValue: 3)
print(c.methodName()) // Prints: 9

var s = StructureImplementingProtocol(someParameter: 2)
print(s.methodName()) // Prints: 6
s.iAmAllowedToModifyThisInstance(withValue: 3)
print(s.methodName()) // Prints: 9

```

Above example is very dry and mostly shows syntax. Examples in subsection **Benefits of using protocols – horizontal relationship** of this chapter allows you to feel what protocols are for.

A matter of initializer

As you know, implementing a protocol initializer on a conforming class, you must mark the initializer implementation with the **required** modifier. You may wonder what to do if a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol? In such a case mark the initializer implementation with *both* the **required** and **override** modifiers:

```

protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // Initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance
    // "override" from SomeSuperClass
    required override init() {
        // Initializer implementation goes here
    }
}

```

```
}  
}
```

Protocols as types

Although protocols don't provide any real functionality themselves, they are a fully-fledged type you can use in your code. In consequence, **you can use protocols in many places where other types are allowed:**

- you can use protocols as a parameter type or return type in a function, method, or initializer;
- you can use protocols as the type of a constant, variable, or property;
- you can use protocols as the type of items in an array, dictionary, or other container.

You can use the **is** and **as** operators to check for protocol conformance, and to cast to a specific protocol. Doing this follows the same, well known, syntax as for type:

- The **is** operator returns *true* if an instance conforms to a protocol and returns *false* if it doesn't.
- The **as?** version of the downcast operator returns an optional value of the protocol's type, and this value is **nil** if the instance doesn't conform to that protocol.
- The **as!** version of the downcast operator forces the downcast to the protocol type and triggers a runtime error if the downcast doesn't succeed.

```

protocol SimpleProtocol1 {
    var property1: Int { get }
}

protocol SimpleProtocol2 {
    var property2: Int { get }
}

class TestClass1: SimpleProtocol1 {
    var property1 = 1
}

class TestClass2: SimpleProtocol2 {
    var property2 = 2
    // You can use protocols as the type of a constant,
    // variable, or property
    var property22: SimpleProtocol1?
}

class TestClass1_2: SimpleProtocol1, SimpleProtocol2 {
    var property1 = 1
    var property2 = 2
}

// You can use protocols as a parameter type
// or return type in a function.
func someFunction(
    object obj: SimpleProtocol1
) -> SimpleProtocol2 {
    let o = TestClass2();
    o.property2 = obj.property1
    o.property22 = obj
    return o
}

let o1 = TestClass1()
let o2 = TestClass1_2()
let x: Any = o1

if (x is SimpleProtocol1){
    let o = someFunction(object: x as! SimpleProtocol1)

    print(o.property2)
    // Prints:
    // 1
    print((o as! TestClass2).property22?.property1 ?? "default")
    // Prints:
    // 1
}

// You can use protocols as the type of items in an array,
// dictionary, or other container.

```

```
var arr = [SimpleProtocol1]()
arr.append(o1)
arr.append(o2)

print(arr)
// Prints:
// [test.TestClass1, test.TestClass1_2]
```

Benefits of using protocols – horizontal relationship

This subsection is the essence of protocols existence depicted on the image at the beginning of this section.

In the [Section 3: Type checking and casting](#) of [Chapter 8: Structures, inheritance and errors handling](#) I've discussed a *polymorphism* topic: a situation when you can treat an object being and instance of a given class as an object of different class which is its ancestor.

EXAMPLE 1

Consider the following hierarchy of objects:

```
class A {}
class B: A {}
class C: A {}
class D: B {}

class E {}
class F: E {}
class G: E {}
```

Now you can create a data structure (array) to store instances of every class from selected family, for example **A**:

```
var familyA = [A]()

familyA.append(A())
familyA.append(B())
familyA.append(C())
familyA.append(D())
```


or pass to a function:

```
func doSomething(on: A) {}

doSomething(on: A())
doSomething(on: B())
doSomething(on: C())
doSomething(on: D())
```

If you try to use object from different family you will get an errors:

```
// Error:
// No exact matches in call to instance method 'append'
familyA.append(E())

// Error:
// Cannot convert value of type 'E' to expected argument type
// 'A'
doSomething(on: E())
```

With protocols you can define relationships among families. Or maybe better: inheritance defines *families*, while protocol defines *friendship*.

```
protocol P1 {}
protocol P2 {}

class A {}
class B: A {}
class C: A, P2 {}
class D: B, P1 {}

class E {}
class F: E, P1, P2 {}
class G: E, P1 {}
```

Now you can create a data structure (array) to store instances of every class from **any** family – the only requirement is to conform selected protocol, for example **P1**:

```
var friends = [P1]()

friends.append(D()) // D from family A
friends.append(F()) // F from family E
friends.append(G()) // G from family E
```

```
func doSomething(on: P2) {}

doSomething(on: C()) // C from family A
doSomething(on: F()) // F from family E
```

If you try to use object which doesn't conform specified protocol, you will get an errors:

```
// Error:
// No exact matches in call to instance method 'append'
friends.append(C())

// Error:
// Argument type 'E' does not conform to expected type 'P2'
doSomething(on: E())
```

EXAMPLE 2

This example is less abstract and is for you to *feel* what protocols are for.

Imagine that you have two families of classes, similarly to a case depicted at the beginning of this chapter, where blue and orange family is showed. Let the first family represents things:

```
class Thing {
}

class Book: Thing {
    var title: String
    var commonlyUsedTitle: String

    init (title: String, commonlyUsedTitle: String) {
        self.title = title
        self.commonlyUsedTitle = commonlyUsedTitle
    }
}
```

The second family is *totally different* and represents living beings:

```
class LivingObject {
```

```

}

class Human: LivingObject {
    var firstName: String
    var secondName: String

    init (firstName: String, secondName: String) {
        self.firstName = firstName
        self.secondName = secondName
    }
}

```

Things and living beings have not too much in common, but every object of these types can be shortly characterize – you can say, it can shortly "introduce itself" saying what kind of object it is. This is a common feature (and very possibly the only one feature) shared among all objects belonging either to inanimate or to living family. Now you can introduce a new class, name it **Common**, and make it a parent for **Thing** and **LivingObject** only to be able to give the ability to introduce to all classes. Making this you admit that **Thing** and **LivingObject** belongs to one super family, which probably is not what you want.

Protocols offer different approach, allowing **Thing** and **LivingObject** preserve their independence. You can think about protocols as an agreement: every class, no matter what family it belongs to, conforming to this protocols agrees to have "inside" all properties and functions defined by protocol. In this example you can define protocol

Introduceable:

```

protocol Introduceable {
    var fullName: String { get }
    var shortName: String { get }
    func introduceYourself() -> String
}

```

Now you may change your classes to conform to this protocol:

```

class Book: Thing, Introduceable {
    var title: String

```

```

var commonlyUsedTitle: String

var fullName: String {
    get {
        title
    }

    set(newValue) {
        title = newValue
    }
}

var shortName: String {
    get {
        commonlyUsedTitle
    }

    set(newValue) {
        commonlyUsedTitle = newValue
    }
}

func introduceYourself() -> String {
    """
    Hello, I'm a book.
    My title is "\($fullName)",
    but you can call me "\($shortName)".
    """
}

init (title: String, commonlyUsedTitle: String) {
    self.title = title
    self.commonlyUsedTitle = commonlyUsedTitle
}
}

class Human: LivingObject, Introduceable {
    var firstName: String
    var secondName: String

    var fullName: String {
        get {
            firstName + " " + secondName
        }
    }

    var shortName: String {
        get {
            String(secondName.prefix(6))
                .lowercased()
                .folding(options: .diacriticInsensitive,
locale: .current)

```

```

    }
}

func introduceYourself() -> String {
    """
    Hello, I'm a human.
    My full name is "\({fullName})",
    but you can call me "\({shortName})".
    """
}

init (firstName: String, secondName: String) {
    self.firstName = firstName
    self.secondName = secondName
}
}

```

At this moment your situation is different. **Book** and **Human** are unrelated classes but thanks to conforming **Introduceable** protocol you can be sure that you can safely call **introduceYourself()** method on every instance of one of these classes. If so, now you can without a problem perform the following code:

```

func introduceObject(object: Introduceable) {
    print(object.introduceYourself())
}

```

on some **Book** and **Human** instances:

```

var b = Book(title: "The C Programming Language",
commonlyUsedTitle: "ANSI C")
var h = Human(firstName: "Piotr", secondName: "Fulmański")

introduceObject(object: b)
introduceObject(object: h)

```

You will see the following output:

```

Hello, I'm a book.
My title is "The C Programming Language",
but you can call me "ANSI C".
Hello, I'm a human.
My full name is "Piotr Fulmański",
but you can call me "fulman".

```

Swift protocol composition

From previous subsection you know that you can specify type of objects allowed to be used based on the protocol adopted by class or structure. But what with a situation, when you want to specify that admissible objects must conform to *more* than one protocol? In that case, you can:

- create an intermediate, temporary type to combine various requirements;
- use *protocol composition*;
- use protocol inheritance (see next subsection).

Protocol composition, as it name states, is the process to combine multiple protocols into a single protocol. You can think about this like defining a temporary protocol that has the combined requirements of all the protocols in the composition (something you can do on your own with protocol inheritance).

Defining protocol composition, you can list as many protocols as you want, separating them using the ampersand & character. Additionally, you can specify one class type, which allows you to specify a superclass.

```
protocol P1 {}  
protocol P2 {}
```

```
class A {}
```

```
class B: A {}  
class C: A, P1 {}  
class D: A, P2 {}  
class E: P1, P2 {}  
class F: A, P1, P2 {}
```

```
func doSomething(on: A & P1 & P2) {}
```

```
// Error:  
// Argument type 'B' does not conform to expected type 'P1'  
//doSomething(on: B())
```

```
// Error:
// Argument type 'C' does not conform to expected type 'P2'
//doSomething(on: C())
// Error:
// Argument type 'D' does not conform to expected type 'P1'
//doSomething(on: D())
// Error:
// Cannot convert value of type 'E' to expected argument type
// 'A'
//doSomething(on: E())
doSomething(on: F())

var container = [A & P1 & P2]()

// Error:
// No exact matches in call to instance method 'append'
//container.append(B())
//container.append(C())
//container.append(D())
//container.append(E())
container.append(F())
```

You can achieved the same result with intermediate type. The following code is almost identical with the above except four places marked with left arrow **<---** and bolded font:

```
protocol P1 {}
protocol P2 {}

class A {}

class Composition: A, P1, P2 {}           // <---

class B: A {}
class C: A, P1 {}
class D: A, P2 {}
class E: P1, P2 {}
class F: Composition {}                   // <---

func doSomething(on: Composition) {} // <---

// Error:
// Cannot convert value of type 'B' to expected argument
// type 'Composition'
//doSomething(on: B())
doSomething(on: F())

var container = [Composition]()         // <---
```

```
// Error:  
// No exact matches in call to instance method 'append'  
//container.append(B())  
container.append(F())
```

Protocols inheritance

A protocol can inherit one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
protocol ProtocolCommon {  
    var mustBeSettable: Int { get set }  
    var maybeSettable: Int { get }  
  
    func methodName() -> Int  
  
    init(someParameter value: Int)  
}  
  
protocol ProtocolForClass2: ProtocolCommon {  
    func iAmAllowedToModifyThisInstance(withValue value: Int)  
}  
  
protocol ProtocolForStructure2: ProtocolCommon {  
    mutating func iAmAllowedToModifyThisInstance(withValue  
value: Int)  
}  
  
class ClassImplementingProtocol2: ProtocolForClass2 {  
    var mustBeSettable: Int = 0  
    var maybeSettable: Int = 0  
  
    func methodName() -> Int {  
        return mustBeSettable + maybeSettable  
    }  
  
    func iAmAllowedToModifyThisInstance(  
        withValue value: Int  
    ) {  
        mustBeSettable = value  
        maybeSettable = value * 2  
    }  
  
    required init(someParameter value: Int) {
```



```

    mustBeSettable = value
    maybeSettable = value * 2
}
}

struct StructureImplementingProtocol2: ProtocolForStructure2 {
    var mustBeSettable: Int = 0
    var maybeSettable: Int = 0

    func methodName() -> Int {
        return mustBeSettable + maybeSettable
    }

    func iAmAllowedToModifyThisInstance(
        withValue value: Int
    ) {
        mustBeSettable = value
        maybeSettable = value * 2
    }

    init(someParameter value: Int) {
        mustBeSettable = value
        maybeSettable = value * 2
    }
}

var c2 = ClassImplementingProtocol2(someParameter: 2)
print(c2.methodName())
// Prints: 6
c2.iAmAllowedToModifyThisInstance(withValue: 3)
print(c2.methodName())
// Prints: 9

var s2 = StructureImplementingProtocol2(someParameter: 2)
print(s2.methodName())
// Prints: 6
s2.iAmAllowedToModifyThisInstance(withValue: 3)
print(s2.methodName())
// Prints: 9

```

You can use protocol inheritance instead of protocol composition. The following code is similar to the composition code presented in preceding subsection except four places marked with left arrow **<--** and bolded font:

```

protocol P1 {}
protocol P2 {}

class A {}

```

```

protocol Composition: A, P1, P2 {}    // <---

class B: A {}
class C: A, P1 {}
class D: A, P2 {}
class E: P1, P2 {}
class F: A, Composition {}           // <---

func doSomething(on: Composition) {} // <---
doSomething(on: F())

var container = [Composition]()      // <---
container.append(F())

```

Default protocol implementation

Protocols can be *extended*. This way you provide method, initializer, subscript, and computed property implementations to conforming types.

For example, if you know that protocol require some method (`toString` below), then you can extend it to provide a method (`prettyPrint` below) which uses the result of the required method:

```

protocol P {
    func toString() -> String
}

extension P {
    func prettyPrint() -> String {
        return "*** \("\(toString())" ***"
    }
}

class A: P {
    var v: Int?

    func toString() -> String {
        guard v != nil else {return "UNDEFINED"}
        return "\("\(v!)"
    }
}

var x = A()

```

```

print(x.prettyPrint())
// Prints:
// *** UNDEFINED ***

x.v = 5
print(x.prettyPrint())
// Prints:
// *** 5 ***

```

As you can see, by creating an extension on the protocol, all conforming types *automatically* gain this method implementation without any additional modification.

If you look at the above code carefully, you will notice that function provided in extension doesn't have to use protocol's required method. It can be any function you want and by implementing it you provide it to any conforming type. Among others, this can be used to provide a *default implementation* to any method or computed property requirement of that protocol. If a conforming type provides its own implementation of a required method or property, that implementation will be used instead of the one provided by the extension.

```

protocol P {
    func important()
}

extension P {
    func important() {
        print("Default implementation")
    }
}

class A: P {}

class B: P {
    func important() {
        print("Improved implementation")
    }
}

var x = A()
x.important()
// Prints:
// Default implementation

```

```
var y = B()  
y.important()  
// Prints:  
// Improved implementation
```


Structs and inheritance

As you know from previous chapter, inheritance only applies to classes. However it is possible to get inheritance-like behavior also for structures if you use protocols.

BASE SOLUTION WITH CLASSES

First, consider very basic hierarchy of classes:

```
class RightPyramid {
    let volume: Double
    let height: Double

    init(params: [String:AnyObject]) {
        height = params["height"] as! Double
        let baseArea = params["baseArea"] as! Double
        volume = 1.0 / 3.0 * baseArea * height
    }
}

class RightSquarePyramid: RightPyramid {
    let edgeB: Double

    override init(params: [String:AnyObject]) {
        edgeB = params["edgeB"] as! Double
        let area = edgeB * edgeB
        var extendedParams = params
        extendedParams["baseArea"] = area as AnyObject
        super.init(params: extendedParams)
    }
}

class RightTrianglePyramid: RightPyramid {
    let baseB: Double
    let heightB: Double

    override init(params: [String:AnyObject]) {
```

```

        baseB = params["baseB"] as! Double
        heightB = params["heightB"] as! Double
        let area = 0.5 * baseB * heightB
        var extendedParams = params
        extendedParams["baseArea"] = area as AnyObject
        super.init(params: extendedParams)
    }
}

func printInfo(pyramid: RightPyramid) {
    print("This is:")
    if let p = pyramid as? RightSquarePyramid {
        print("a right square pyramid:")
        print("base edge length=\(p.edgeB)")
    } else if let p = pyramid as? RightTrianglePyramid {
        print("a right triangle pyramid:")
        print("base triangle base=\(p.baseB)")
        print("base triangle height=\(p.heightB)")
    }
    print("height=\(pyramid.height)\nvolume=\(pyramid.volume)")
}

var params = ["baseB": 10.0,
              "heightB": 5.0,
              "height": 2.0]
as [String : AnyObject]
let pyramid = RightTrianglePyramid(params: params)
printInfo(pyramid: pyramid)
// Prints:
// This is:
// a right triangle pyramid:
// base triangle base=10.0
// base triangle height=5.0
// height=2.0
// volume=16.666666666666664

```

Now you may ask if there is possibility to implement this hierarchy but using structures. Answer is positive, although no solution is perfect.

SOLUTION 1

In this approach you use composition of value types.

```

struct RightPyramid {
    let volume: Double
    let height: Double

    init(params: [String:AnyObject]) {
        height = params["height"] as! Double
    }
}

```



```

    let baseArea = params["baseArea"] as! Double
    volume = 1.0 / 3.0 * baseArea * height
}
}

struct RightSquarePyramid {
    let rightPyramid: RightPyramid
    let edgeB: Double

    init(params: [String:AnyObject]) {
        edgeB = params["edgeB"] as! Double
        let area = edgeB * edgeB
        var extendedParams = params
        extendedParams["baseArea"] = area as AnyObject
        rightPyramid = RightPyramid(params: extendedParams)
    }
}

struct RightTrianglePyramid {
    let rightPyramid: RightPyramid
    let baseB: Double
    let heightB: Double

    init(params: [String:AnyObject]) {
        baseB = params["baseB"] as! Double
        heightB = params["heightB"] as! Double
        let area = 0.5 * baseB * heightB
        var extendedParams = params
        extendedParams["baseArea"] = area as AnyObject
        rightPyramid = RightPyramid(params: extendedParams)
    }
}

func printInfo(pyramid: Any) {
    func printCommonInfo(pyramid: RightPyramid) {
        print("height=\(pyramid.height)\nvolume=\(pyramid.volume)")
    }
    print("This is:")
    if type(of: pyramid) == RightSquarePyramid.self {
        if let p = pyramid as? RightSquarePyramid {
            print("a right square pyramid:")
            print("base edge length=\(p.edgeB)")
            printCommonInfo(pyramid: p.rightPyramid)
        }
    } else if type(of: pyramid) == RightTrianglePyramid.self {
        if let p = pyramid as? RightTrianglePyramid {
            print("a right triangle pyramid:")
            print("base triangle base=\(p.baseB)")
            print("base triangle height=\(p.heightB)")
            printCommonInfo(pyramid: p.rightPyramid)
        }
    }
}
}

```

```

}

var params = ["baseB": 10.0,
             "heightB": 5.0,
             "height": 2.0]
    as [String : AnyObject]
let pyramid = RightTrianglePyramid(params: params)
printInfo(pyramid: pyramid)
// Prints:
// This is:
// a right triangle pyramid:
// base triangle base=10.0
// base triangle height=5.0
// height=2.0
// volume=16.666666666666664

```

Drawback of this solution is that you *violate the encapsulation principle*, exposing the internal composition to the outside world.

`RightPyramid` becomes a property of compound type (`RightSquarePyramid` and `RightTrianglePyramid`):

```

struct RightSquarePyramid {
    let rightPyramid: RightPyramid
    // Some code goes here
}

struct RightTrianglePyramid {
    let rightPyramid: RightPyramid
    // Some code goes here
}

```

Although you have `RightSquarePyramid` type you don't treat it as a type but rather as a composition of other types because to get an access to `volume` and `height` you have to "manually" extract them from property of `RightPyramid` type.

SOLUTION 2

In this approach you use a protocol and one intermediate structure (`RightPyramidData`):

```

protocol RightPyramid {
    // Error:
    // Protocols cannot require properties to be immutable;

```

```

// declare read-only properties by using 'var' with
// a '{ get }' specifier
// let volume: Double
var volume: Double { get }
var height: Double { get }
}

private struct RightPyramidData: RightPyramid {
    let volume: Double
    let height: Double

    init(params: [String:AnyObject]) {
        height = params["height"] as! Double
        let baseArea = params["baseArea"] as! Double
        volume = 1.0 / 3.0 * baseArea * height
    }
}

struct RightSquarePyramid: RightPyramid {
    // BEGIN: To conform protocol
    let volume: Double
    let height: Double
    // END: To conform protocol
    let edgeB: Double

    init(params: [String:AnyObject]) {
        edgeB = params["edgeB"] as! Double
        let area = edgeB * edgeB
        var extendedParams = params
        extendedParams["baseArea"] = area as AnyObject
        let rightPyramidData = RightPyramidData(
            params: extendedParams
        )
        volume = rightPyramidData.volume
        height = rightPyramidData.height
    }
}

struct RightTrianglePyramid: RightPyramid {
    // BEGIN: To conform protocol
    let volume: Double
    let height: Double
    // END: To conform protocol
    let baseB: Double
    let heightB: Double

    init(params: [String:AnyObject]) {
        baseB = params["baseB"] as! Double
        heightB = params["heightB"] as! Double
        let area = 0.5 * baseB * heightB
        var extendedParams = params
        extendedParams["baseArea"] = area as AnyObject
    }
}

```

```

    let rightPyramidData = RightPyramidData(
        params: extendedParams
    )
    volume = rightPyramidData.volume
    height = rightPyramidData.height
}

func printInfo(pyramid: RightPyramid) {
    print("This is:")
    if let p = pyramid as? RightSquarePyramid {
        print("a right square pyramid:")
        print("base edge length=\(p.edgeB)")
    } else if let p = pyramid as? RightTrianglePyramid {
        print("a right triangle pyramid:")
        print("base triangle base=\(p.baseB)")
        print("base triangle height=\(p.heightB)")
    }
    print("height=\(pyramid.height)\nvolume=\(pyramid.volume)")
}

var params = ["baseB": 10.0,
              "heightB": 5.0,
              "height": 2.0]
    as [String : AnyObject]
let pyramid = RightTrianglePyramid(params: params)
printInfo(pyramid: pyramid)
// Prints:
// This is:
// a right triangle pyramid:
// base triangle base=10.0
// base triangle height=5.0
// height=2.0
// volume=16.666666666666664

```

Notice that with this approach both `printInfo(pyramid:)` and code following it is exactly of the same shape as it is given in [Base solution with classes](#). Drawback is that it results in code verbosity (notice existence of intermediate `RightPyramidData` structure) and manual property manipulation at lines:

```

let rightPyramidData = RightPyramidData(
    params: extendedParams
)
volume = rightPyramidData.volume
height = rightPyramidData.height

```

SOLUTION 3

This solution shows how you can get rid of intermediate structure using protocol extension to provide default data extracting implementation.

```
protocol RightPyramid {
    var volume: Double { get }
    var height: Double { get }
}

extension RightPyramid {
    static func parseFields(
        params: [String:AnyObject]
    ) -> (Double, Double) {
        let height = params["height"] as! Double
        let baseArea = params["baseArea"] as! Double
        let volume = 1.0 / 3.0 * baseArea * height

        return (volume, height)
    }
}

struct RightSquarePyramid: RightPyramid {
    // BEGIN: To conform protocol
    let volume: Double
    let height: Double
    // END: To conform protocol
    let edgeB: Double

    init(params: [String:AnyObject]) {
        edgeB = params["edgeB"] as! Double
        let area = edgeB * edgeB
        var extendedParams = params
        extendedParams["baseArea"] = area as AnyObject
        (volume, height) = RightSquarePyramid.parseFields(
            params: extendedParams
        )
    }
}

struct RightTrianglePyramid: RightPyramid {
    // BEGIN: To conform protocol
    let volume: Double
    let height: Double
    // END: To conform protocol
    let baseB: Double
    let heightB: Double

    init(params: [String:AnyObject]) {
        baseB = params["baseB"] as! Double
        heightB = params["heightB"] as! Double
    }
}
```

```

    let area = 0.5 * baseB * heightB
    var extendedParams = params
    extendedParams["baseArea"] = area as AnyObject
    (volume, height) = RightSquarePyramid.parseFields(
        params: extendedParams
    )
}

func printInfo(pyramid: RightPyramid) {
    print("This is:")
    if let p = pyramid as? RightSquarePyramid {
        print("a right square pyramid:")
        print("base edge length=\(p.edgeB)")
    } else if let p = pyramid as? RightTrianglePyramid {
        print("a right triangle pyramid:")
        print("base triangle base=\(p.baseB)")
        print("base triangle height=\(p.heightB)")
    }
    print("height=\(pyramid.height)\nvolume=\(pyramid.volume)")
}

var params = ["baseB": 10.0,
              "heightB": 5.0,
              "height": 2.0]
as [String : AnyObject]
let pyramid = RightTrianglePyramid(params: params)
printInfo(pyramid: pyramid)
// Prints:
// This is:
// a right triangle pyramid:
// base triangle base=10.0
// base triangle height=5.0
// height=2.0
// volume=16.666666666666664

```

As you can see, **RightPyramidData** structure is not needed any more and manual property manipulation is replaced by on-line call:

```

(volume, height) = RightSquarePyramid.parseFields(
    params: extendedParams
)

```


Generics

With *generics* you can write very flexible and reusable code without duplicating it only because you want it to work with different type(s). In simple words, generic is an adjustable stamp or template you can use to *generate* identical code with some minor type changes. Generics help you to stop repeating yourself.

Generic functions

Generic functions can work with any type. Any time you have a functionality *common for different types* you can write a universal (generic) code where instead of real type a *placeholder type* is used. In Swift a placeholder type, known as a *type parameter*, specify its name, and is written immediately after the function's name, between a pair of matching angle brackets (such as `<T>` for `T` type parameter). The generic version of the function uses a placeholder type name (`T`, in this case) instead of an actual type name (such as `Int`, or `String`). The actual type to use in place of `T` is determined every time a function is used.

```
func arrayReverse<T>(_ array: inout [T]){
    let count = array.count
    var temp: T
    for i in 0..
```

This generic code, when called with an array of **Strings**:

```
var arrayString = ["one", "two", "three", "four", "five"]
print(arrayString)
// Prints:
// ["one", "two", "three", "four", "five"]

arrayReverse(&arrayString)
```

behind a scene will result with the following autogenerated code:

```
func arrayReverse(_ array: inout [String]) { // Here is a
                                                // change 1 of 2
    let count = array.count
    var temp: String // Here is a change 2 of 2

    for i in 0..
```

which will do the job:

```
print(arrayString)
// Prints:
// ["five", "four", "three", "two", "one"]
```

On the other hand, when called with array of **Ints**:

```
var arrayInt = [1, 2, 3, 4, 5]
print(arrayInt)
// Prints:
// [1, 2, 3, 4, 5]

arrayReverse(&arrayInt)
```

behind a scene will result with the following autogenerated code:

```
func arrayReverse(_ array: inout [Int]) { // Here is a
                                                // change 1 of 2
```

```

let count = array.count
var temp: Int // Here is a change 2 of 2

for i in 0..

```

which will do the job for **Ints** this time:

```

print(arrayInt)
// Prints:
// [5, 4, 3, 2, 1]

```

Type constraints in generic functions

Although generics functions can work with any type, it's sometimes useful to enforce certain type constraints on the types that can be used. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition. Type constraints concerns generic functions as well as generic types (see next subsection).

```

func functionName<T: className, U: protocolName>(
    argT: T, argU: U
) {
    // Function body goes here
}

```

In the code below it is safe to call **toString** function on **object** because you can be sure that **Type** conforms to **P** protocol requiring this function:

```

protocol P {
    func toString() -> String
}

struct A: P {
    var v: Int
}

```

```

    func toString() -> String {
        "\({v})"
    }
}

struct B {
    var v: Int
}

// It is safe to call toString on object because
// you can be sure that Type conforms to P protocol
func prettyPrint<Type: P>(object: Type) {
    print("*** \({object.toString()}) ***")
}

var x = A(v: 5)
prettyPrint(object: x)

var y = B(v: 5)
// Error:
// Global function 'prettyPrint(object:)' requires that 'B'
// conform to 'P'
//prettyPrint(object: y)

```

Generic types

Generic types are maybe even more often used than generic functions. The most basic examples are quite natural: basic data structures like arrays, dictionaries, stacks or queues. As an example, take a look at a very basic implementation of a priority queue:

```

struct PriorityQueue<T> {
    struct Item<T> {
        var item: T
        var priority: Int
    }

    var items = [Item<T>]()

    mutating func push(_ item: T, withPriority priority: Int) {
        items.append(Item(item: item, priority: priority))
    }

    mutating func pop() -> T? {
        guard items.count > 0 else {
            return nil
        }
    }
}

```

```

    }

    var highestPriorityIndex = 0
    var highestPriorityValue = items[0].priority

    for index in 1..

```

If you run it, you will see:

```

one
two
three
four
five
undefined

```

On the other hand the following code:

```

var pqI = PriorityQueue<Int>()

pqI.push(55, withPriority: 5)

```

```

pqI.push(22, withPriority: 2)
pqI.push(11, withPriority: 1)
pqI.push(44, withPriority: 4)
pqI.push(33, withPriority: 3)

print(pqI.pop() ?? "undefined")
print(pqI.pop() ?? "undefined")
print(pqI.pop() ?? "undefined")
print(pqI.pop() ?? "undefined")
print(pqI.pop() ?? "undefined")
print(pqI.pop() ?? "undefined")

```

will print:

```

55
44
33
22
11
undefined

```

Extending a generic types

When extending a generic type, you don't provide a type parameter list as part of the extension's definition. Instead, the type parameter list from the original type definition is available within the body of the extension, and the original type parameter names are used to refer to the type parameters from the original definition.

Based on the previous code, you can add an extension to it:

```

extension PriorityQueue {
  var itemWithHighestPriority: T? {
    guard items.count > 0 else {
      return nil
    }

    var highestPriorityIndex = 0;
    var highestPriorityValue = items[0].priority;

    for index in 1..

```

```

    }
}

return items[highestPriorityIndex].item
}
}

pq.push("five", withPriority: 5)
pq.push("two", withPriority: 2)
pq.push("one", withPriority: 1)

print(pq.itemWithHighestPriority ?? "undefined")

```

If you run it, you will see:

one

Generic protocols: associated types

When defining a protocol, it's sometimes useful to declare one or more *associated types* as part of the protocol's definition. This is a way you make generic protocols. An associated type gives a placeholder name to a type that is used as part of the protocol. The actual type to use for that associated type isn't specified until the protocol is adopted. Associated types are specified with the `associatedtype` keyword.

```

protocol Resettable {
    associatedtype ItemType
    mutating func reset(
        toValue value: ItemType, withPriority priority: Int
    )
}

```

Now you can use the extended protocol either in nongeneric way:

```

struct PriorityQueueString: Resettable {
    struct Item {
        var item: String
        var priority: Int
    }

    var items = [Item]()
}

```

```

mutating func push(
    _ item: String,
    withPriority priority: Int
) {
    items.append(Item(item: item, priority: priority))
}

mutating func pop() -> String? {
    guard items.count > 0 else {
        return nil
    }

    var bestPriorityIndex = 0;
    var bestPriorityValue = items[0].priority;

    for index in 1..

```



```
print(pqs.pop() ?? "undefined")
// Prints: two
```

or in generic way:

```
struct PriorityQueueConformingToProtocols<T>: Resettable {
    struct Item<T> {
        var item: T
        var priority: Int
    }

    var items = [Item<T>]()

    mutating func push(_ item: T, withPriority priority: Int) {
        items.append(Item(item: item, priority: priority))
    }

    mutating func pop() -> T? {
        guard items.count > 0 else {
            return nil
        }

        var highestPriorityIndex = 0;
        var highestPriorityValue = items[0].priority;

        for index in 1..

```

```
pqp.reset(toValue: "none", withPriority: 1)
print(pqp.pop() ?? "undefined")
// Prints: none

print(pqp.pop() ?? "undefined")
// Prints: none

print(pqp.pop() ?? "undefined")
// Prints: two
```

Note, that in generic case we don't have to use `typealias` keyword.

Miscellaneous topics

You will learn:

- How to use pattern matching with **case-let**.

This is not the end

This is the end of this book but I hope not your last adventure with Swift. Reading and practicing knowledge you have gained so far, you will have understanding of most basic Swift's "building blocks". Starting from variables, through extensions and ending on protocols and generics, now you are ready to try to make a code on your own or try something else like making app for iOS.

You know a lot but there is still even more to discover. In this chapter I will show you few concepts which you don't have to know at this moment but which show how broad Swift is. It is simply to stimulate your curiosity and encourage you to get to know it on your own discovering new features and areas of application.

case-let pattern

The keyword `case` used in other than `switch` statements looks at first sight strange and awkward. However, once you get used to it, you will never want to throw it away. The key to accept `case-let` is to stop thinking that `case` is inseparable linked with `switch`. Rather, see it as a situation where `case` is used by `switch`. If so, why other statements, like `if`, `guard` or `for`, may not use it as well?

To make this chapter's contents clear you may find helpful quick refreshment of [Section 2: switch - case statement](#) from [Chapter 5: Tuples, switch and extensions](#), where the following complex pattern matching example summarizes `switch-case` syntax:

```
var point2D: (Double, Double)

point2D = (2.5, 2)

switch point2D {
case (0, 0):
    print("Origin")
case (let x, 0):
    print("Point \(x),0) is on the OX axis")
case (0, let y):
    print("Point 0, \(y) is on the OY axis")
case let (x, y) where x > y:
    print("Point \(x), \(y) from a 2D subspace")
default:
    print("Eeee...")
}
// Prints:
// Point (2.5, 2.0) from a 2D subspace
```

As you can see, inside `switch-case` you can bind values to constants or variables (with `let` or `var` keyword). You can also specify additional condition with `where` keyword.

Frequently `switch-case` is used with enum types ([Chapter 3: Arrays and enumerations, Section: 2 Enumerations](#)):

```
enum Action {
    case turnLeftDegree (Double),
        turnRightDegree (Double),
        makeForwardSteps (Int),
        makeBackwardSteps (Int),
        saySomething (String)
}

var currentAction = Action.makeForwardSteps(10)

switch currentAction {
case .turnLeftDegree(let degree):
    print("Turn left by \(degree) degree")
case let .turnRightDegree(degree):
    print("Turn right by \(degree) degree")
case .makeForwardSteps(let steps):
    print("Make \(steps) step(s) forward")
case let .makeBackwardSteps(steps):
    print("Make \(steps) step(s) backward")
case let .saySomething(text):
    print("Say: \(text)")
}
// Prints:
// Make 10 step(s) forward
```

The ability to *pattern matching* in some sense redefines `switch-case` statements and transforms it into really very handy tool. This is so attractive that introducing *pattern matching* ability in other *condition-like* statements shouldn't surprise you.

Generally speaking, the `case let x = y` pattern, wether is used as a part of `if`, `guard` or even `for`, allows you to check if `y` does *match the pattern x*. If you remember this, syntax will not scare you.

IF-CASE-LET

```
if case let .makeForwardSteps(steps) = currentAction {  
    print("Current action matches moving forward with \ (steps)  
steps")  
}  
// Prints:  
// Current action matches moving forward with 10 steps
```

An alternate form:

```
if case .makeForwardSteps(let steps) = currentAction {  
    print("Current action matches moving forward with \ (steps)  
steps")  
}  
// Prints:  
// Current action matches moving forward with 10 steps
```

IF-CASE-LET-WHERE

You can combine the `if-case-let` with a *condition* separating them with a comma (,):

```
if case let .makeForwardSteps(steps) = currentAction,  
    steps > 20 {  
    print("Current action matches moving forward with more than  
20 steps (with \ (steps) steps)")  
} else {  
    print("No match or fail at condition check")  
}  
// Prints:  
// No match or fail at condition check
```

You can specify more than one condition:

```
if case let .makeForwardSteps(steps) = currentAction,  
    steps > 5, steps < 30 {  
    print("Current action matches moving forward with \ (steps)  
steps, and this is the value in the expected range (5, 30)")  
} else {  
    print("No match or fail at condition check")  
}  
// Prints:  
// Current action matches moving forward with 10 steps,  
// and this is the value in the expected range (5, 30)
```

GUARD-CASE-LET[-WHERE]

```
guard case let .makeForwardSteps(steps) = currentAction,
    steps > 10 else {
    print("No match or fail at condition check")
    return
}

print("Proceed with action matches moving forward with \(steps)
steps")

// Prints:
// No match or fail at condition check
```

FOR-CASE-LET[-WHERE]

Combining **for** and **case** can also let you iterate on a collection conditionally. Using **for-case-let** is semantically similar to using a **for** loop and wrapping its whole body in an **if-case** block: in result it will only iterate and process the elements that match the pattern.

```
var actions = [Action]()
actions.append(.makeForwardSteps(2))
actions.append(.turnLeftDegree(5))
actions.append(.makeForwardSteps(50))
actions.append(.turnLeftDegree(2))
actions.append(.makeForwardSteps(100))
actions.append(.makeForwardSteps(10))

for case let Action.makeForwardSteps(steps) in
    actions
where
    steps > 45
{
    print("Long step ahead: \(steps) steps")
}

// Prints:
// Long step ahead: 50 steps
// Long step ahead: 100 steps
```

FOR-WHERE

Note that **for** without the **case** pattern matching part but preserving **where** part is also a valid Swift syntax:

```
for number in [1, 2, 3, 4, 5]
where
    number % 2 == 0
{
    print("\(number), ", terminator: "")
}
// Prints:
// 2, 4,
```


Bibliography

[ARD] Arduino

1. `constrain()`, retrieved 2021-03-30,
<https://www.arduino.cc/reference/en/language/functions/math/constrain/>
2. `map()`, retrieved 2021-03-30,
<https://www.arduino.cc/reference/en/language/functions/math/map/>

[IUO] Implicitly Unwrapped Optional

1. *Abolish ImplicitlyUnwrappedOptional type*, retrieved 2021-03-17,
<https://github.com/apple/swift-evolution/blob/master/proposals/0054-abolish-iuo.md>
2. *Reimplementation of Implicitly Unwrapped Optionals*, retrieved 2021-03-17,
<https://swift.org/blog/iuo/>

[JAVADOC] Java documentation

1. *The finally Block*, retrieved 2021-04-20,
<https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html>

[SD] Swift documentation

1. Property Wrappers section in *Properties*, retrieved 2021-04-08,
<https://docs.swift.org/swift-book/LanguageGuide/Properties.html>

2. Audrey Tam, *SwiftUI Property Wrappers*, retrieved 2021-05-21, <https://www.raywenderlich.com/21522453-swiftui-property-wrappers>

[SELF] Self and self

1. *Self and self in Swift*, retrieved 2021-03-30, <https://learnappmaking.com/self-swift-how-to/>
2. *Self vs self - what's the difference?*, retrieved 2021-03-30, <https://www.hackingwithswift.com/example-code/language/self-vs-self-whats-the-difference>

[TM] Value type

1. *Reference vs. Value Types in Swift*, retrieved 2021-04-01, <https://www.raywenderlich.com/9481-reference-vs-value-types-in-swift>

