Hide menu

Course Introduction

Enumerations

Video: Introduction to enumerations
6 min

Reading: Enumerations with raw and associated values examples
15 min

Reading: Exercise: Work with raw and associated values
30 min

Reading: Solution: Work with raw and associated values
10 min

Practice Quiz: Self review: Work with raw and associated values
9 min

Practice Quiz: Knowledge check: Enumerations
21 min

Reading: Additional resources
5 min

What are sets?

Enumerations with raw and associated values examples

So far, you've learned that enums are commonly used to group related code into a single definition. This reading will teach you more about enums and how to use enums with raw and associated values. You'll also learn more about enum methods and how to use the **`where`** Swift keyword to write a program with fewer lines of code, making your code cleaner and easier to understand.

By reviewing the examples in this reading, you'll be able to work with enums and understand how to assign values to the end of enum cases.

Enums with raw values

Raw values attach predefined values of the same type to enum cases. To assign a raw value to an enum, you need to declare the type of raw value you want to use for the enum. This can be achieved by inserting the desired data type in the enum signature.

```
enum PastaTypes: Int { }
```

Next, you'll be able to add raw values to each of the enum cases using a simple swift syntax like the values associated with a variable or a constant. To demonstrate, review the enum example below that represents four common types of pasta including spaghetti, penne, ravioli, and rigatoni. The enum is called **`PastaTypes`** and consists of four cases. Currently, no additional information is assigned to any of the cases.

```
enum PastaTypes: Int {
  case spaghetti
  case penne
  case ravioli
  case rigatoni
}

```

Next, let's assign raw values to indicate the cooking time required for each type of pasta. This will allow the person cooking the pasta to know the exact time in minutes when the pasta is "Al Dente"(cooked just enough to keep its texture). First, the type is specified in the enum signature as type **`Int`**. Then, each case is assigned a unique value using the assignment operator. Remember that each of the raw values associated with the enum case must be unique.

```
enum PastaTypes: Int {
  case spaghetti = 8
  case penne = 10
  case ravioli = 11
  case rigatoni = 12
}
```

Your first step when adding raw values to an enum is to choose the data type for your raw values. Aligning your raw values in this way is common practice in Swift programming and ensures that you have a clear and clean code structure.

To access the raw values, you can create a variable and assign a raw value of the enum case. Using this variable, you can print a message to the console and check the cooking time for the desired pasta type.

```
let cookingTime = PastaTypes.spaghetti.rawValue
print("Spaghetti will be perfectly cooked in \(cookingTime) minutes.")

```

You can also create a function called **`cookingPerfectPasta`** and pass the enum as the argument. The function will print the type of pasta and cooking time associated with the exact **`PastaType`** case. This will help you test other enum cases.

```
func cookingPerfectPasta(pasta: PastaTypes) {
  let cookingTime = pasta.rawValue
  print("\(pasta) will be perfectly cooked in \(cookingTime) minutes.")
}
cookingPerfectPasta(pasta: .rigatoni)

```

CaseIterable Protocol

Before you explore how enums work with associated values, let's briefly demonstrate the power of using the `CaseIterable` protocol with enumeration. Don't worry too much about what protocols are, you'll learn more about that soon. For now, all you need to know is that the `CaseIterable` protocol enables you to iterate over the cases in an enum.

To conform to the `CaseIterable` protocol, add the name of the protocol after the data type in the enum signature.

```
enum PastaTypes: Int, CaseIterable {
  case spaghetti
  case penne
  case ravioli
  case rigatoni
}

```

Conforming to the `CaseIterable` enables you to use the `allCases` property. This property is essentially a regular array and can be used to check the total enum cases. This can be done using the `.count` property.

Let's demonstrate this with the following example:

```
let totalCaseCount = PastaTypes.allCases.count
print(totalCaseCount)

```

Running this code in the console will return "4" as the total number of cases in the `PastaTypes` enum.

Enums with associated values

In the first example, you learned how using raw values can be helpful while working with enums. Now you will focus on using associated values. Associated values allow you to pass additional information to an enum. To assign an associated value to an enum, you don't need to specify a data type to the enum. Instead, you specify the data type for a variable in each of the enum cases.

To demonstrate, let's create another enum called `PastaTypesA` with the same pasta cases. The newly created enum will work with associated values.

```
enum PastaTypesA {
  case spaghetti(cookingTime: Int)
  case penne(cookingTime: Int)
  case ravioli(cookingTime: Int)
  case rigatoni(cookingTime: Int)
}
```

In the example, the enum also consists of four cases representing four types of pasta. The main difference is that it declares that additional information is associated with the enum cases. The associated values will represent the cooking time for each type of pasta.

Let's demonstrate how to use the new enum with associated values.

```
var checkIfCooked = PastaTypesA.spaghetti(cookingTime: 8)
```

The code illustrates that a new variable called `checkIfCooked` is created and assigned a value. The assigned value is the `spaghetti` enum case from the `PastaTypesA` enum, and the cooking time is set to 8 minutes.

Next, let's create a condition that prints a custom message based on the `cookingTime` value.

```
if cookingTime < 8 {
  print("Spaghetti is still not fully cooked...")
} else {
  print("Spaghetti is cooked, take it out of the water!")
}
```

You may agree that writing the above code for each enum case may be time-consuming. To combat this, you can use functions.

Using enum inside a function

Using an enum inside a function allows you to write more succinct and expressive code. For instance, you can use a function to evaluate whether the pasta is cooked or if it needs more time. Since there are several pasta types that would need to be evaluated, a function allows you to streamline that process.

Explore the example below where the function employs a switch statement that switches on an enum case:

```
func checkIfCooked(for pasta: PastaTypesA) {
```

```
  switch pasta {
  case .spaghetti(let cookingTime):
    if cookingTime < 8 {
      print("Spaghetti is still not fully cooked...")
    } else {
      print("Spaghetti is cooked, take it out of the water!")
    }

  default: return
  }
}
```

Next, let's call the function and pass a parameter value.

```
checkIfCooked(for: .spaghetti(cookingTime: 9))
```

The function call will produce the following result:
“Spaghetti is cooked, take it out of the water!”

While the function improves the overall code readability, there's additional room for improvement. Using an abundance of `if` and `else` statements inside a switch statement can clutter your code, making it difficult to read and work with. As a solution, the `where` keyword can be used to simplify your code while achieving the same desired results.

Using "where" statements

The `where` statement checks for the condition where the condition is true. Suppose you print a message to the console where the cooking time for the spaghetti pasta is greater than or equal to 8 minutes. The `where` keyword can be used to evaluate conditions in a single line, as compared to `if else` statements that run across several lines.

To demonstrate this, let’s create another method to check if the pasta is properly cooked. Note that a new name is used to ensure the code compiles as intended.

```
func checkIfCooked2(for pasta: PastaTypesA) {
  switch pasta {
  case .spaghetti(let cookingTime) where cookingTime >= 8:
    print("Spaghetti is cooked, take it out of the water!")
  default: print("Pasta is not cooked.")
  }
}
checkIfCooked2(for: .spaghetti(cookingTime: 7))
```

The function receives the same arguments as the one created before and switches on the enum case too. The only difference is that the `where` keyword checks the condition. If the condition is met, the message is printed to the console.

Concluding thoughts

By completing this reading, you’ve learned how to assign values to enum cases. You explored enum properties and methods and how to use the where keyword to compare enum cases.

As a result, you should now be more comfortable using enumeration with raw and associated values.

Mark as completed