15時間でわかる Swift 集中講座

解答と解説

解答と解説

1時間目　確認テスト

Q1

変数はvarを使用して宣言します。

```
var Numbers = 5
```

Q2

型推論での例とデータ型を指定した書式を使用する例を示します。

```
// 型推論での例
var osName = ("OS X", "iOS")

// データ型を指定した書式での例
var osName:(String, String) = ("OS X", "iOS")
```

Q3

型推論での例とデータ型を指定した書式を使用する例を示します。

```
var language = ["Swift","Objective-C","Java"]

// データ型を指定した書式での例
//var language:[String] = ["Swift","Objective-C","Java"]
```

Q4

```
let vegetables = [1:"tomato", 2:"potato", 3:"carrot"]
```

2時間目　確認テスト

Q1

変数noに3, 5, それ以外の値（この例では2）を与えてみましょう。それぞれFizz, Buzz, 代入した値が表示されます。

```
let no = 3  // Fizzが表示される
// let no = 5   // Buzzが表示される
// let no = 2   // 2が表示される

switch (no % 3, no % 5) {
case (0, 0):
    print("FizzBuzz")
case (0, _):
    print("Fizz")
case (_, 0):
    print("Buzz")
default:
    print(no)
}
```

Q2

以下は変数xに6を、yに3を与えた場合の解答例です。xを5よりも小さい値、yを3よりも小さい値にも変更して実行してみましょう。

```
var x: Int = 6
var y: Int = 3

if (x >= 5 && y >= 3) {
    print("Swift is cool!!")
}
```

解答と解説

Q3

```
var ans : Int = 0

for num in 1...10 {
    //print("現在\(i)回目の繰り返しです")
    ans += num
}

print("1～10までの加算結果は\[1](ans)です")
```

3時間目　確認テスト

Q1

nilが代入可能な変数を宣言するには、データ型の後ろに?を付けます。

```
var myMessage: String? = nil
```

Q2

アンラップをするには変数の後ろに!を付けます。

```
var myMessage: String? = nil

myMessage = "私の名前は"

print(myMessage! + " " + "HIROです")
```

Q3

関数名をGetAreaとした例を示します。

```
func GetArea(tate: Int, yoko: Int) {
    print(tate * yoko)
}

GetArea(3, yoko: 5)
```

Q4

タプルを返す関数は、戻り値の型にタプルを指定します。

関数calcの内部では、最大値、最小値、平均値を計算しています。Swiftにはあらかじめ最大値を求める関数maxと最小値を求める関数が備わっていますが、引数を2つしか取らないため、2重にして3つの値の最大値や最小値を求めています。平均値は四則演算で求めています。最後に、求めた3つの値をタプルとして返しています。

```
func calc(x1: Int, x2: Int, x3: Int) -> (Int, Int, Int) {
    let maxNum = max(max(x1, x2), x3)   // 最大値を計算
    let minNum = min(min(x1, x2), x3)   // 最小値を計算
    let avgNum = (x1 + x2 + x3) / 3     // 平均値を計算

    return (maxNum, minNum, avgNum)
}

// 関数calcを実行
var ans = calc(5, x2: 7, x3: 3)

print(ans)
```

4時間目　確認テスト

Q1

2つの値を足し算するクロージャを変数addNumに代入する例を示します。

```
var addNum = { x1,x2 -> Int in x1 + x2 }
```

Q2

Q1で作成したクロージャを実行する例を示します。

```
print(addNum(4, 7))
```

Q3

列挙型DayOfWeekの定義例を示します。

```
enum DayOfWeek {
    case Sun, Mon, Tue, Wed, Thu, Fri, Sat
}
```

解答と解説

Q4 列挙型GenderにRaw値を指定する例を示します。データ型としてIntを指定しています。また、オートインクリメントの機能を使用してRaw値を指定するので、最初のSunのみに値を設定しています。

```
enum DayOfWeek : Int {
    case Sun = 1
    case Mon
    case Tue
    case Wed
    case Thu
    case Fri
    case Sat
}
```

Q5 文字列「I like swift.」から「I love swift」に変更する例を示します。
はじめに「swift」を削除し、「love」を挿入しています。

```
var myMessage = "I like swift."

// swiftの部分の範囲を取得
let range = myMessage.startIndex.advancedBy(2)..<myMessage.startIndex.
advancedBy(6)

// "swift"の文字列を削除
myMessage.removeRange(range)

// "love"を挿入
myMessage.insertContentsOf("love".characters, at: myMessage.startIndex.
advancedBy(2))
```

5時間目　確認テスト

Q1

Rectangleクラスの作成例を示します。

Heightを保持型プロパティでWidthを計算型プロパティで定義しています。

Widthプロパティに値が代入される場合はsetブロックが実行されます。5未満の場合かどうかはif文で判断し、強制的に5をセットしています。

```
class Rectangle {
    var Height: Int = 0
    var _width: Int = 0

    var Width: Int {
        get {
            return _width;
        }
        set (newWidth) {
            if newWidth < 5 {
                _width = 5
            } else {
                _width = newWidth
            }
        }
    }
}
```

Q1

Rectangleクラスにイニシャライザと GetAreaメソッドを定義する例を示します。

GetAreaメソッドでは、クラス内で管理している _widthとHeightを使用して長方形の面積を計算して返しています。

```
class Rectangle {
    var Height: Int = 0
    var _width: Int = 0

    init () {
```

```
        self.Height = 3
        self._width = 10
    }

    func GetArea() -> Int {
        return self._width * self.Height
    }

    // WidthプロパティはQ1を参照
}
```

Rectangleクラスの使用例を示します。

Rectangleクラスのインスタンスを生成語Widthプロパティに7、Heightプロパティに3を代入しています。また、GetAreaメソッドで求めた面積を表示しています。

```
var rect = Rectangle()

rect.Width = 7
rect.Height = 3

print(rect.GetArea())
```

6時間目　確認テスト

Q1

サブスクリプトの定義例を示します。

表データ（縦×横）のデータを管理するサブスクリプトです。

イニシャライザは表の大きさとなる行数（rowCnt）と列数（colCnt）を受け取り、データを格納する配列サイズを設定します。

サブスクリプトの定義では、行番号と列番号を受け取ってデータの取得と設定を行うことができるようにしています。

データを登録する場合はNumbersData[1, 0] = "020-0000" のようにします。

```
class Numbers {
    let rowCount: Int
    let colCount: Int
    var data: [String]

    init(rowCnt: Int , colCnt: Int) {
        self.rowCount = rowCnt
        self.colCount = colCnt
        data = Array(count: self.rowCount * self.colCount,
repeatedValue: "")
    }

    subscript(rowNo: Int, colNo: Int) -> String {
        get {
            return data[(rowNo * colNo) + colNo]
        }
        set {
            data[(rowNo * colNo) + colNo] = newValue
        }
    }
}

var NumbersData = Numbers(rowCnt: 5, colCnt: 3)
```

```
// データの代入
NumbersData[0, 0] = "郵便番号"
NumbersData[0, 1] = "住所"
NumbersData[0, 2] = "氏名"
NumbersData[1, 0] = "020-0000"
NumbersData[1, 1] = "東京都"
NumbersData[1, 2] = "阿部"
NumbersData[2, 0] = "123-4567"
NumbersData[2, 1] = "沖縄県"
NumbersData[2, 2] = "具志堅"

// 住所データを表示
print("1つ目の住所:\(NumbersData[1,1])")
print("2つ目の住所:\(NumbersData[2,1])")
```

Carクラスの定義例を示します。
各プロパティは保持型プロパティで定義しています。

```
class Car {
    var Wheel : Int = 0     // ハンドルを表すプロパティ
    var breaks  : Int = 0   // ブレーキを表すプロパティ
    var Name : String = ""  // 名前
}
```

Carクラスを継承したTruckクラスの定義例を示します。
継承をする場合はクラス名の後ろに「: 継承元クラス名」を記述します。

```
// Carクラスを継承したTruckクラス
class Truck: Car {
    var CarryingPlatform : String = ""
}
```

Q4

エクステンションを使用してクラスを拡張する例を示します。

キーワードextensionを使用して定義し、内部にはクラス持たせるメンバの定義を記述します。

```
extension Truck {
    func GetName() -> String {
        return "名前は\(self.Name)です"
    }
}
```

7時間目　確認テスト

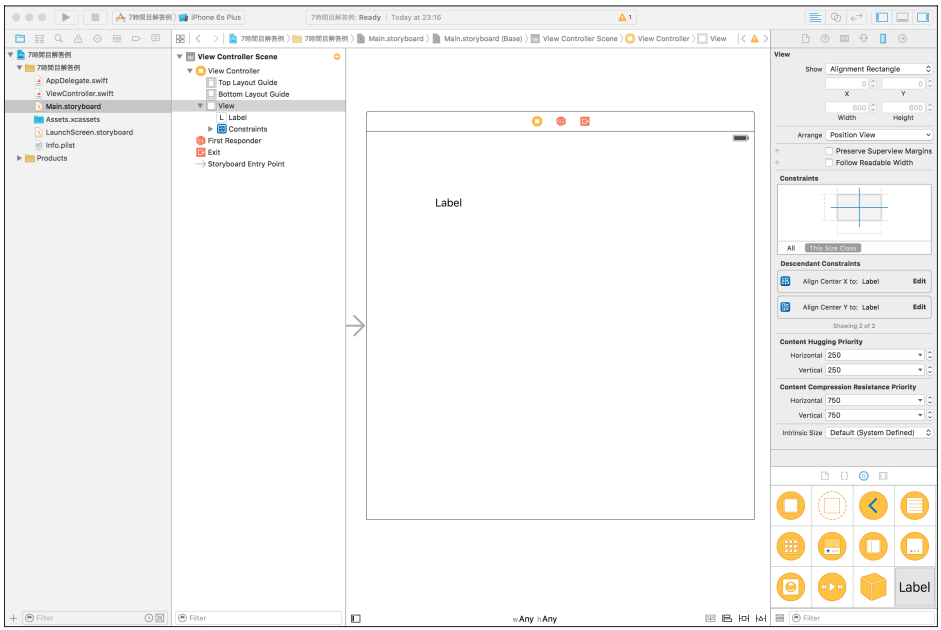
Q1

新規でSingle View Applicationを作成する方法は、「7-1　プロジェクトの作成とワークスペース」を参照ください。

Q2

Labelの配置例を示します。

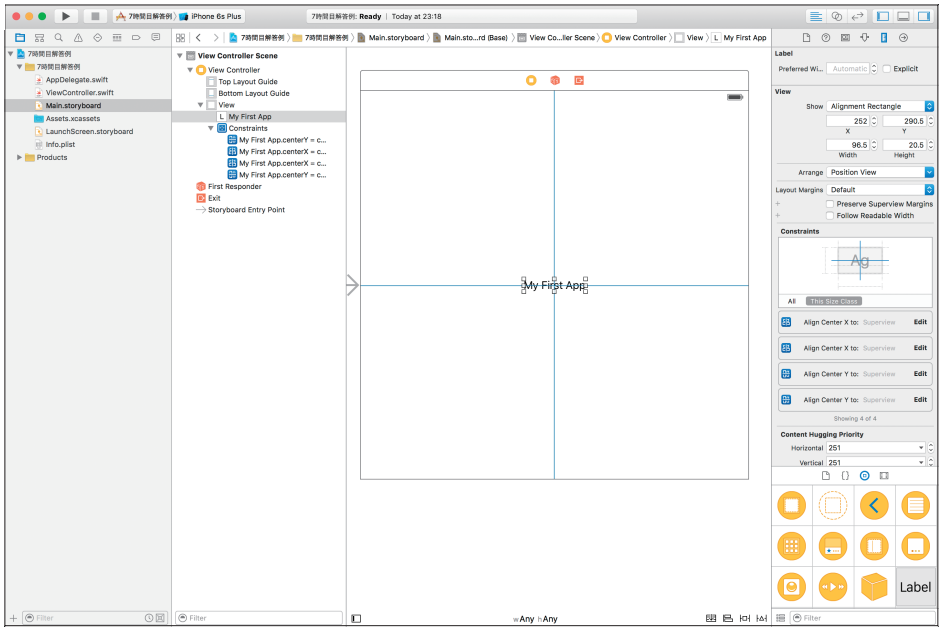
画面にLabelを貼り付け後、水平方向と垂直方向の中心に表示されるように制約を付けています。



解答と解説

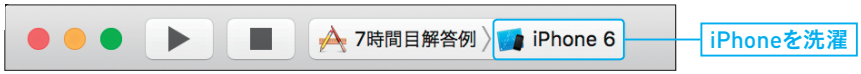
Q3

Labelの文字列を「My First App」に変更し、中央に配置した例を示します。



Q4

シミュレータをiPhone6にするには下図ようにします。



Q5

シミュレータは、既定では縦方向に表示されます。
回転した場合の例を示します。

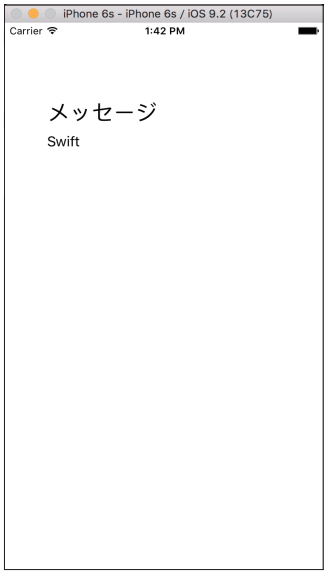
8時間目　確認テスト

Q1

Q2

Q2

Q1～Q3を通して作成したアプリケーションの実行例を示します。



コード例は以下の通りです。

Q1のアウトレット接続については「8-1-2 アウトレット接続を作成する」を参照してください。

Q2のフォントの変更については「8-2-3 フォント」を参照してください。

Q3のコードからのラベル生成については「8-3 UI部品の動的生成」を参照してください。この例では、UILabel生成時に表示位置も設定しています。

```
class ViewController: UIViewController {

    // Q1の解答例
    // 画面に貼り付けた「メッセージ」というLabelのアウトレット接続
    @IBOutlet weak var labelMessage: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Q2の解答例
```

解答と解説

```
        // 画面に貼り付けた「メッセージ」ラベルのフォントを
        // Times New Roman, サイズ24にします
        labelMessage.font = UIFont(name: "Times New Roman", size: 26)

        // Q3の解答例
        let myLabel: UILabel =  UILabel(frame: CGRectMake(50, 120, 130,
50))

        myLabel.text = "Swift"

        self.view.addSubview(myLabel)

    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

9時間目　確認テスト

Q1

Text Fieldで発生したイベント名をLabelに表示する例を示します。

「9-1-3 デリゲートを使用した処理」を参考にしてください。

新規で「Single View Application」を作成したら、画面にLabelとText Fieldを1つずつ貼り付けて、アウトレット接続を作成します（**表9-1**、**表9-2**）。

表9-1 Labelのアウトレット接続

設定項目	設定値
Connection	Outlet
Name	labelTextFieldEvent
Type	UILabel

表9-2 Text Fieldのアウトレット接続

設定項目	設定値
Connection	Outlet
Name	myText
Type	UITextField

コードは以下のように作成します。

```
class ViewController: UIViewController, UITextFieldDelegate {   ・・・@

    @IBOutlet weak var labelTextFieldEvent: UILabel!
    @IBOutlet weak var myText: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()

        self.myText.delegate = self   ・・・A
    }

    // 編集開始時   ・・・B
    func textFieldDidBeginEditing(textField: UITextField) {
```

```
        labelTextFieldEvent.text = "textFieldDidBeginEditing"
    }

    // クリアボタン押下時　・・・C
    func textFieldShouldClear(textField: UITextField) -> Bool {
        labelTextFieldEvent.text = "textFieldShouldClear"

        return true
    }

    // returnキー押下時　・・・D
    func textFieldShouldReturn(textField: UITextField) -> Bool {
        labelTextFieldEvent.text = "textFieldShouldReturn"

        // きーボードを閉じる
        myText.resignFirstResponder()

        return true
    }


    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

@でViewControllerクラスにUITextFieldDelegateプロトコルを適用しています。

Aでデリゲート設定をし、B、C、Dで画面に配置したText Fieldで発生したイベントの処理を実施しています。

実行例を以下に示します。

Q2

UIStepperを使用して、5ずつカウントアップするアプリケーションの作成例を示します。

画面にUIStepperとLabelを1つずつ配置したら、以下の**表10-1**～**10-3**の通りアウトレット接続とアクション接続を作成します。

表10-1 Labelのアウトレット接続

設定項目	設定値
Connection	Outlet
Name	labelCount
Type	UILabel

表10-2 Labelのアウトレット接続

設定項目	設定値
Connection	Outlet
Name	stepper
Type	UIStepper

表10-3 Labelのアウトレット接続

設定項目	設定値
Connection	Action
Name	stepChanged
Type	UIStepper
Event	Value Changed

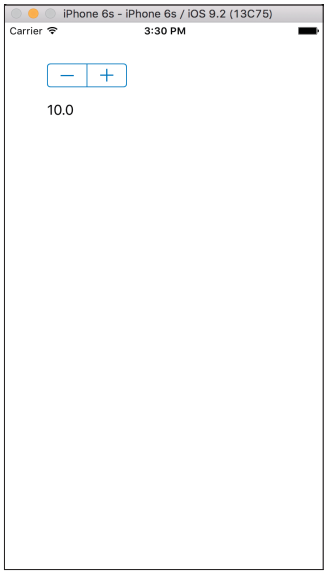
続いて、画面上でUI Stepperを選択して、アトリビュートインスペクタでStepperのStepを「5」に設定しておきます。

以上が設定できたら、コードを入力しましょう。

コード例を以下に示します。

```
class ViewController: UIViewController {

    @IBOutlet weak var labelCount: UILabel!
    @IBOutlet weak var stepper: UIStepper!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically
from a nib.
    }

    // Stepperのカウントアップ/ダウン時の処理   ・・・@
    @IBAction func stepChanged(sender: UIStepper) {
        // UIStepperのカウント値を表示
        labelCount.text = stepper.value.description   ・・・A
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

@がUI Stepperでカウントの変更があった場合に実行されるアクション接続です。
AでUI Stepperの値を取得してラベルに表示しています。
実行例を以下に示します。



Q3

Segmentd ControlでDateが選択された場合と、Timeが選択された場合とで書式を変更する例を示します。
ここでは「9-6-1　日付の表示」で作成したアプリケーションを改造して作成します。

```
class ViewController: UIViewController {

    @IBOutlet weak var myDatePicker: UIDatePicker!
    @IBOutlet weak var labelDate: UILabel!

    var format: String = "yyyy/MM/dd"   ・・・@

    override func viewDidLoad() {
        super.viewDidLoad()
    }
```

```
    @IBAction func valueChanged(sender: AnyObject) {
        switch sender.selectedSegmentIndex {
        case 0: // 【Time】選択時
            myDatePicker.datePickerMode = UIDatePickerMode.Time

            self.format = "HH:mm"   ・・・ A

        case 1: // 【Date】選択時
            myDatePicker.datePickerMode = UIDatePickerMode.Date

            self.format = "yyy/MM/dd"   ・・・ B

        default: break

        }

        labelDate.text = toDateString(myDatePicker.date, style: self.
format)   ・・・ C
    }


    @IBAction func dateChanged(sender: UIDatePicker) {
        labelDate.text = toDateString(myDatePicker.date, style: self.
format)   ・・・ D
    }

    // 省略
}
```

Q3に対するコードを作成するには、Segmented Contrlで「Time」「Date」切り替え時に書式が変更されるようにします。

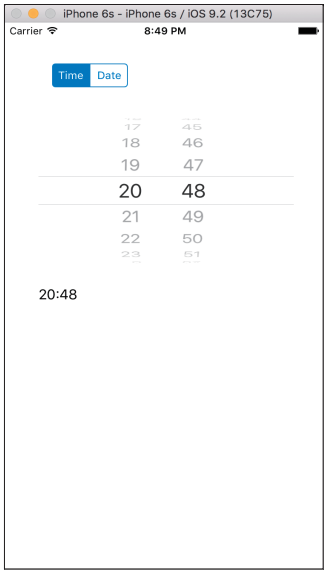
@でformatというString型の変数を準備し、あらかじめDateが選択されたときの書式を代入しておきます。

valueChangedのメソッドでは「Time」が選択されたときにformat変数に書式「HH:mm」をセットし（ A)、「Date」が選択されたときに「yyyy/MM/dd」をセットしています（ B）

あとはswitch文を抜けたときに Cで書式を設定しています。

またdateChnagedメソッドも変更しています。こちらも、「Time」「Date」に合わせて書式が変更されるようにしています。

実行例を以下に示します。



10時間目　確認テスト

Q1

ここでは「9-3 Slider」で作成したアプリケーションに保存機能を付加する例を示します。

各スライダーで設定した色の値を読み書きできるようにします

コード例は以下の通りです。

```
class ViewController: UIViewController {

    @IBOutlet weak var sliderRed: UISlider!
    @IBOutlet weak var sliderGreen: UISlider!
    @IBOutlet weak var sliderBlue: UISlider!

    let defaults = NSUserDefaults.standardUserDefaults()  ・・・@
    // 色の値を保持する変数  ・・・A
    var dRed: Float = 1.0
    var dGreen: Float = 1.0
    var dBlue: Float = 1.0

    override func viewDidLoad() {
        super.viewDidLoad()

        // データの読み込み  ・・・B
        dRed = defaults.floatForKey("Red")
        dGreen = defaults.floatForKey("Green")
        dBlue = defaults.floatForKey("Blue")

        // 読み込んだ値をスライダーに反映  ・・・C
        sliderRed.value = dRed
        sliderGreen.value = dGreen
        sliderBlue.value = dBlue

        // 読み込んだ値で画面の色を変更  ・・・D
```

```
        self.view.backgroundColor = UIColor(red: CGFloat(dRed), green:
CGFloat(dGreen), blue: CGFloat(dBlue), alpha: 1.0)
    }

    @IBAction func sliderValueChanged(sender: AnyObject) {
        let redVal:CGFloat = CGFloat(sliderRed.value)
        let greenVal:CGFloat = CGFloat(sliderGreen.value)
        let blueVal:CGFloat = CGFloat(sliderBlue.value)

        self.view.backgroundColor = UIColor(red: redVal, green:
greenVal, blue: blueVal, alpha: 1.0)

        // スライダーの値を保存　・・・E
        defaults.setFloat(sliderRed.value, forKey: "Red")
        defaults.setFloat(sliderGreen.value, forKey: "Green")
        defaults.setFloat(sliderBlue.value, forKey: "Blue")
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

@はアプリケーション内でNSUserDefaultsを使用するための変数を宣言しています。

Aは各スライダーの値を保持する変数を宣言しています。

Bはアプリケーション起動時に、スライダーの各値を読み込む処理です。各色のキーは「Red」「Green」「Blue」としています。

データの読み込み終了後、Cで各スライダーの値を設定し、Dで画面に色を反映させています。

スライダーの値が変更された場合は、sliderValueChangedメソッドの中の Eで値を保存しています。

Q2 プロパティリストに都道府県データを保存してPicker Viewに表示するには、「10-4 プロパティリストを使用する」の登録データを猫から都道府県に変更するだけです。

都道府県データを登録して、Picker Viewに表示されるようにしてみましょう。

11時間目 確認テスト

Q1 SKSceneファイルを作成して、画像やテキストを表示する方法については「11-2 ノードを配置する」を参照してください。

Q2 タッチによるキャラクターの移動については「11-3 タッチ操作」を参照して下さい。自分の好きな画像ファイルに変更してみましょう。

Q3 キャラクターの移動範囲を画面の上半分に固定する例を示します。

ここでは、「11-3-5 キャラクターの移動範囲を限定する」のコードを改造することとします。

リストは以下の通りです。

@はロケットの配置位置を設定しています。今回は縦方向の位置を画面の中心となるように「self.frame.height * 0.5」としています。

AとBでは現在のタッチ位置が画面の上半分かどうかを判断しています。if文の比較演算子をリスト11.14と反対にしています。

```
override func didMoveToView(view: SKView) {
    // 省略

    // キャラクター画像の設定
    rocket.position = CGPoint(x: 100, y: self.frame.height * 0.5 )  ・・・@
    rocket.size = CGSize(width: 50, height: 100)

    addChild(rocket)
}

override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {
    let point = touches.first!.locationInNode(self)
```

```
    // 現在の座標が画面高さの半分よりも上にあるとき　・・・A
    if point.y > self.frame.height * 0.5 {
        rocket.position = point
    }
}

override func touchesMoved(touches: Set<UITouch>, withEvent event:
UIEvent?) {
    let point = touches.first!.locationInNode(self)

    // 現在の座標が画面高さの半分よりも上にあるとき　・・・B
    if point.y > self.frame.height * 0.5 {
        rocket.position = point
    }
}
```

12時間目　確認テスト

Q1

ここでは矢印の画像を読み込み、画面をタッチする毎に移動、回転、拡大、縮小をするアプリケーションの例を示します。

矢印の画像をarrow.pngという名前で作成してプロジェクトに組み込んだ上で以下のコードを入力して下さい。

プロジェクトにはarrow.pngという画像ファイルを含めた上で実行してください。

解答と解説

```
import Foundation
import SpriteKit

class SampleScene : SKScene {

    let arrow = SKSpriteNode(imageNamed: "arrow")  ・・・ @
    var touchCount: Int = 0  ・・・ A

    override func didMoveToView(view: SKView) {
        // 背景画像の設定

        arrow.position = CGPoint(x: self.size.width * 0.5, y: self.size.
height * 0.5)  ・・・ B
        arrow.size = CGSize(width: self.size.width / 4, height: self.
size.height / 4)  ・・・ C

        addChild(arrow)  ・・・ D
    }

    override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {
        let action: SKAction  ・・・ E
        let point = touches.first!.locationInNode(self)  ・・・ F

        if touchCount > 3 {  ・・・ G
            touchCount = 0
        }

        switch touchCount {  ・・・ H
        case 0:
            action = SKAction.moveTo(CGPoint(x: point.x, y: point.y),
duration: 1)
        case 1:
```

```
            action = SKAction.rotateByAngle(CGFloat(M_PI), duration: 1)
        case 2:
            action = SKAction.scaleTo(2, duration: 1)
        default:
            action = SKAction.scaleTo(0.5, duration: 1)
        }

        arrow.runAction(action)  ・・・I

        touchCount++  ・・・J
    }
}
```

＠で矢印の画像の読み込みを行っています。

Aのtouch Count変数はタッチ回数を管理するための変数です。これはタッチの回数によってアクションを切り分ける処理に使用します

画面表示時の処理（didMoveToViewメソッド）では、Bで矢印の画像を読み込み、Cで矢印のサイズを設定し、Dで画面に矢印を表示しています。

画面がタッチされた場合（touchesBeganメソッド）は、はじめにアクションを代入するための変数を準備しています（E）。

Fは、タッチされた位置を取得しています。

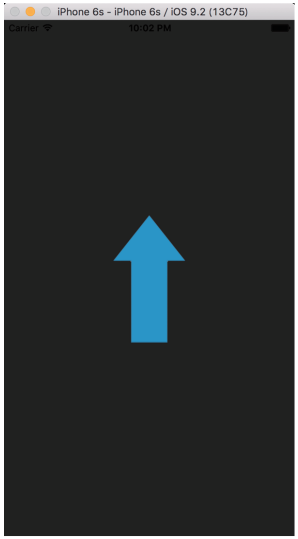
Gは、タッチ回数が3を超えていないかをチェックし、超えている場合は0に戻す処理をしています。これはHのswitchでtouchCountが0～3までの範囲で処理を分岐させるためです。touchCountが0の場合は移動を、1の場合は左に90度の回転を、2の場合サイズ2倍に拡大、その他（3の場合が該当）の場合はサイズを0.5倍に縮小しています。

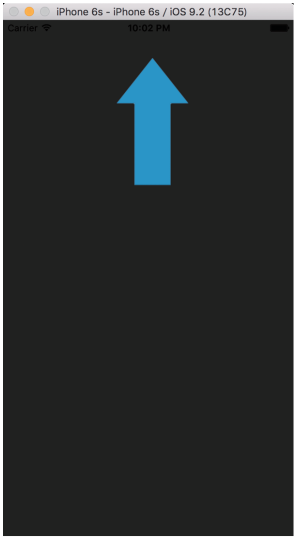
switch文でアクションを作成したら、Iで矢印に設定しています。

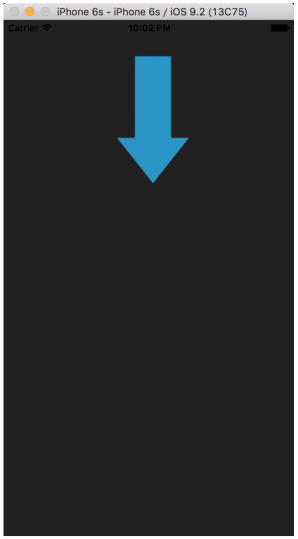
最後に、次のタッチに備えtouchCountをカウントアップしています。

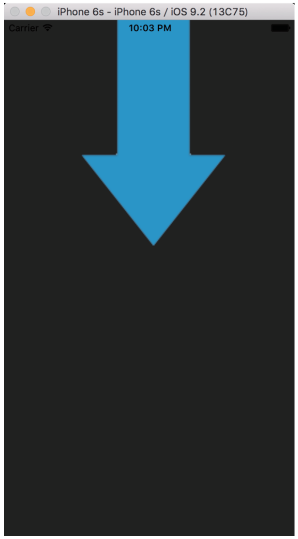
実行例は以下のとおりです。

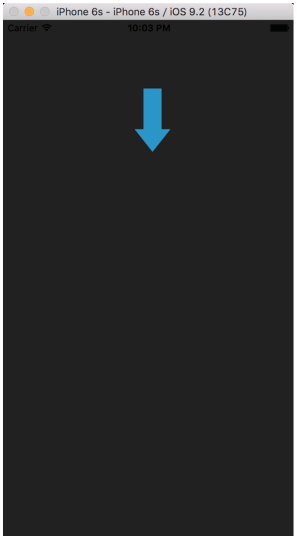
解答と解説


iPhone 6s - iPhone 6s / iOS 9.2 (13C75)
Carrier
10:02 PM


iPhone 6s - iPhone 6s / iOS 9.2 (13C75)
Carrier
10:02 PM


iPhone 6s - iPhone 6s / iOS 9.2 (13C75)
Carrier
10:02 PM


iPhone 6s - iPhone 6s / iOS 9.2 (13C75)
Carrier
10:03 PM


iPhone 6s - iPhone 6s / iOS 9.2 (13C75)
Carrier
10:03 PM

ここではQ1のコードを改造して、アニメーションを作成する例を示します。

画面がタッチされたときに、移動、回転、拡大、縮小を連続して行うアニメーションを作成します。

touchesBeganメソッドのコードのみ示します。didMoveToViewメソッドのコードはQ1を参照してください。

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {
    let point = touches.first!.locationInNode(self)

    let action1 = SKAction.moveTo(CGPoint(x: point.x, y: point.y),
duration: 1)
    let action2 = SKAction.rotateByAngle(CGFloat(M_PI), duration: 1)
    let action3 = SKAction.scaleTo(2, duration: 1)
    let action4 = SKAction.scaleTo(0.5, duration: 1)

    let sequence = SKAction.sequence([action1, action2, action3,
action4])

    // 作成したアニメーションの適用
    arrow.runAction(sequence)
}
```

action1～action4の変数に、移動、回転、拡大、縮小のアクションを代入した後、SKAction.sequeceメソッドを使用して、連続したアニメーションを作成しています。

最後に矢印のrunActionメソッドでアニメーションを実行しています。

ここではQ1のコードを改造して、タッチする度にサウンドが鳴るように処理を追加する例を示します。

サウンドを鳴らす部分のみのコードを掲載します。

追加部分は ＠と Ａです。 ＠でtouch.mp3という音源を読み込み。 Ａで再生しています。

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {
    let action: SKAction
    let point = touches.first!.locationInNode(self)

    let touchSound = SKAction.playSoundFileNamed("touch.mp3",
waitForCompletion: false)  ・・・ @
    arrow.runAction(touchSound)  ・・・ A

    // 省略
}
```

13時間目　確認テスト

Q1

ここでは画面に重力を設定する例を示します。

画面に重力を設定するには、以下のコード例のようにdidMoveToViewメソッド内でself.physicsWorld.gravityに値を設定します。重力の値はCGVectorで作成します。

```
import Foundation
import SpriteKit

class SampleScene : SKScene {

    override func didMoveToView(view: SKView) {
        // 画面への重力設定
        self.physicsWorld.gravity = CGVector(dx: 0, dy: -5)
    }
}
```

Q2

ここでは12時間目の解答例で使用した矢印の画像を落下させる例を示します。

プロジェクトには矢印の画像（arrow.png）を組み込んだ上でコードを入力してください。

```
import Foundation
import SpriteKit

class SampleScene : SKScene {

    override func didMoveToView(view: SKView) {
        // 画面への重力設定
        self.physicsWorld.gravity = CGVector(dx: 0, dy: -5)

        // 矢印画像の読み込み
        let arrowTexture = SKTexture(imageNamed: "arrow")  ・・・ @
        let arrow = SKSpriteNode(texture: arrowTexture)  ・・・ A

        arrow.size = CGSizeMake(self.frame.width * 0.1, self.frame.width
* 0.1)  ・・・ B
        arrow.position = CGPoint(x: self.size.width * 0.5, y: self.size.
height - arrow.size.height)  ・・・ C

        // 矢印への物理ボディの設定
        arrow.physicsBody = SKPhysicsBody(texture: arrowTexture, size:
arrow.size)  ・・・ D

        addChild(arrow)  ・・・ E
    }

}
```

@は矢印の画像を読み込みarrowTextureへ格納しています。

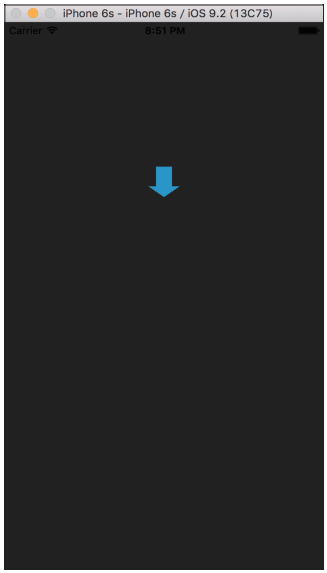
Aは@で読み込んだ画像を使用してSKSpriteNodeを作成しています。

Bは矢印の大きさを、Cは表示位置を設定しています。

矢印を落下させるためにDで物理ボディを与えています。SKPhysicsBodyの第1引数にarrowTextureを与えることで、物理シミュレーションの影響を受ける範囲が矢印の形と同じになります。

Eで画面に矢印を追加しています。

実行例を以下に示します。



Q3

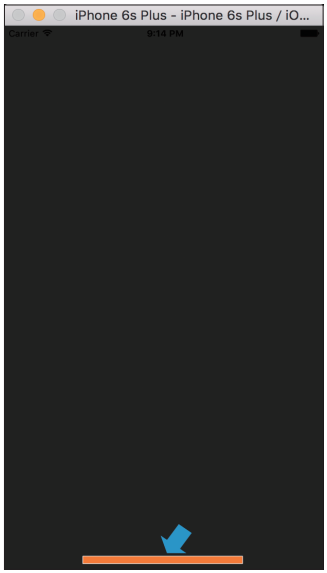
ここではQ2の解答例に地面を追加して、矢印が着地する例を示します。

地面のコード例は、「リスト13.3　地面の作成」と同じです。

Q2のコードの後に、以下を追加してください。解説は317ページを参照してください。

```
// 地面の作成
let ground = SKShapeNode(rect: CGRectMake(0, 0, self.frame.midX, 10))
ground.position = CGPointMake(self.frame.midX - ground.frame.width *
0.5, 10)
ground.fillColor = UIColor.orangeColor()

ground.physicsBody = SKPhysicsBody(rectangleOfSize: ground.frame.size,
    center: CGPointMake(ground.position.x, ground.frame.height * 0.5))

ground.physicsBody?.dynamic = false

addChild(ground)
```

実行例を以下に示します。



矢印は落下すると地面へ着地します。先端が尖っているので、最後は倒れて止まります。

Q4

ここでは、Q1～Q3までのコードにもう1つすり抜ける地面を追加する例を示します。Q3のコードの後ろに以下のコードを入力して下さい。

```
// すり抜ける地面の作成
let ground2 = SKShapeNode(rect: CGRectMake(0, 0, self.frame.midX, 10))
ground2.position = CGPointMake(self.frame.midX - ground2.frame.width *
0.5, self.frame.midY)
ground2.fillColor = UIColor.greenColor()

ground2.physicsBody = SKPhysicsBody(rectangleOfSize: ground2.frame.size,
    center: CGPointMake(ground2.position.x, ground2.frame.height * 0.5))

ground2.physicsBody?.dynamic = false

addChild(ground2)
```

```
// カテゴリビットマスクと衝突ビットマスクの設定
arrow.physicsBody?.categoryBitMask = 0b0001
arrow.physicsBody?.collisionBitMask = 0b0010

ground.physicsBody?.categoryBitMask = 0b0010
ground.physicsBody?.collisionBitMask = 0b0001

ground2.physicsBody?.categoryBitMask = 0b0100
ground2.physicsBody?.categoryBitMask = 0b0100
```

すり抜ける地面は、Q3で入力したコードをコピーして貼り付けground2に変更しています。また表示位置を画面高さの中心位置になるようにしています。

カテゴリビットマスクは矢印が0b0001、画面一番下にある地面が0b0010、画面中心に表示した地面を0b0100としています。

矢印が衝突できる地面は元々あった画面一番下の地面とするために、衝突ビットマスクを0b0010に設定しています。

実行すると、画面中央に表示した地面をすり抜けて、一番下の地面に着地します。

Q5

ここでは、矢印が地面に衝突したときに「衝突」というテキストを表示する例を示します。

Q4のコードを以下のように改造します。

```
class SampleScene : SKScene, SKPhysicsContactDelegate {  ・・・ⓐ

    override func didMoveToView(view: SKView) {
        self.physicsWorld.contactDelegate = self  ・・・A

        // 省略

        // 地面と衝突したときにイベントを発生させる
        ground.physicsBody?.contactTestBitMask = 0b0001  ・・・B
    }

    func didBeginContact(contact: SKPhysicsContact) {
```

```
        let labelNode = SKLabelNode(fontNamed: "Helvetica")  ・・・C

        labelNode.text = "衝突"  ・・・D
        labelNode.fontSize = 20  ・・・E
        labelNode.position = contact.contactPoint  ・・・F
        labelNode.zPosition = 10  ・・・G

        addChild(labelNode)
    }
}
```

物体が衝突したことを受け取るために、didBeginCntactメソッドを発生させる必用があります。そこで @、AでSKPhysicsContactDelegateプロトコルを適用し、衝突イベントを受け取れるようにします。

Bは地面に矢印が衝突したときにdidBeginContactメソッドを発生させるためのコードです。地面（ground）のcontactTestBitMaskには、矢印のカテゴリビットマスクの値0b0001を指定します。

続いてdidBeginContactMaskのコードを確認しましょう。

Cは表示するテキストのフォントをHelveticaに設定しています。

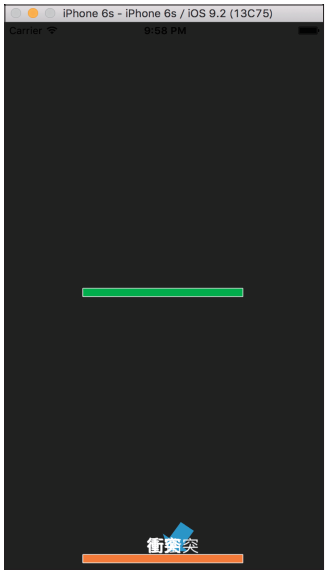
Dは表示するテキスト「衝突」を、Eをテキストのフォントサイズを設定しています。

Fはテキストの表示位置を設定するコードです。衝突位置はcontact.contactPointで取得することができます。

Gは表示する文字が他の物体に隠れないようにするための設定です。

最後に画面に表示しています。

実行例を以下に示します。



矢印がバウンドするので、「衝突」の文字が複数表示されます。

14時間目　確認テスト

Q1

本書「14-2 電子コンパス」では画像を使用して方位を示すアプリケーションを作成しました。

ここでは方位を文字で表示するアプリケーションを作成する例を示します。

尚、CoreLocationフレームワークの設定方法については、「14-1-3 CoreLocationフレームワークとのリンク」を参照してください。

はじめにMain.storyboardの中央に方位を表示するためのラベルを1つ貼り付けてください。

続いて以下のように、ラベルのアウトレット接続を作成します。

表14-1 Lラベルのアウトレット接続

設定項目	設定値
Connection	Outlet
Name	direction
Type	UILabel

コードは以下のように入力します。

```
import UIKit
import CoreLocation
class ViewController: UIViewController, CLLocationManagerDelegate {

    @IBOutlet weak var direction: UILabel!

    var locationMgr: CLLocationManager

    required init?(coder aDecoder: NSCoder) {
        locationMgr = CLLocationManager()

        super.init(coder: aDecoder)!
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        locationMgr.delegate = self   ・・・ @
        locationMgr.headingFilter = 10   ・・・ A
        locationMgr.headingOrientation = CLDeviceOrientation.Portrait

        locationMgr.startUpdatingHeading()
    }

    func locationManager(manager: CLLocationManager, didUpdateHeading
newHeading: CLHeading) {

        let rt = newHeading.magneticHeading   ・・・ B

        switch rt {   ・・・ C
        case 0..<22.5:
                direction.text = "北"
```

```
        case 22.5..<67.5:
                direction.text = "北東"
        case 67.5..<112.5:
                direction.text = "東"
        case 112.5..<157.5:
            direction.text = "南東"
        case 157.5..<202.5:
            direction.text = "南"
        case 202.5..<247.5:
            direction.text = "南西"
        case 247.5..<292.5:
            direction.text = "西"
        case 292.5..<337.5:
            direction.text = "北西"
        default:
            direction.text = "北"
        }

    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

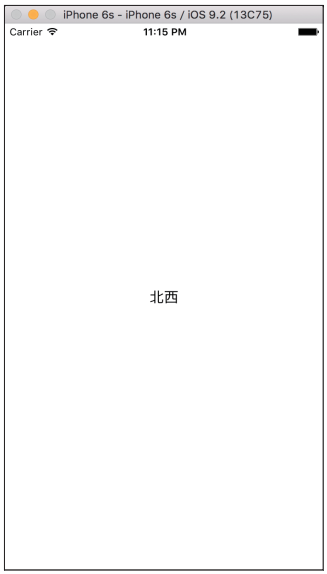
viewDidLoadとlocationManagerのコードについて説明します。その他については本書を参照して下さい。

@ではデリゲートの設定を行い、角度の変更通知を受け取れるようにしています。

Aは10度動く毎に通知を受け取るようにする設定です。

Bは通知があった場合の角度を取得してrtに代入しています。

最後にCのswitch文で、方位の範囲（360度を8分割しています）と比較して文字を表示しています。



Q2

ここでは「14-3 加速度センサー」のサンプルコードを改造して、ボールを転がすアプリケーションを作成する例を示します。

本サンプルコードを作成する場合は、プロジェクトにCoreMotionを組み込んでください。また、ball.pngをAssets.xcassetsフォルダに配置してください。

コード例は以下の通りです。

```
import SpriteKit
import CoreMotion

class SampleScene : SKScene {

    var myMotionManager: CMMotionManager!
    var speedX: Double = 0   ・・・ @
    var speedY: Double = 0   ・・・ A

    override func didMoveToView(view: SKView) {
        // 画面の中心位置   ・・・ B
        let centerX = self.frame.width / 2
        let centerY = self.frame.height / 2
```

```
        // ボールの作成　・・・C
        let ballTexture = SKTexture(imageNamed: "ball")
        let ball = SKSpriteNode(texture: ballTexture)
        ball.size = CGSize(width: 50, height: 50)
        ball.position = CGPointMake(centerX, centerY)
        addChild(ball)

        // MotionManagerを生成
        myMotionManager = CMMotionManager()
        // 1/60秒ごとにデータを取得する
        myMotionManager.accelerometerUpdateInterval = 1 / 60　・・・D

        // 加速度データの取得を実行
        let accelerometerHandler = { (data: CMAccelerometerData?, error:
NSError?) -> Void in　・・・E
            // 加速度センサーからの値をスピード値とする　・・・F
            self.speedX += data!.acceleration.x
            self.speedY += data!.acceleration.y

            // 現在の位置にスピード値を加算して新しい位置を求める　・・・G
            var posX: CGFloat = ball.position.x + CGFloat(self.speedX)
            var posY: CGFloat = ball.position.y + CGFloat(self.speedY)

            // ボールの半分のサイズを求める
            let ballHalfSize = ball.frame.width / 2

            if posX < 0.0 + ballHalfSize {　・・・H
                posX = 0.0 + ballHalfSize
                self.speedX *= -0.5
            } else if posX > self.size.width - ballHalfSize　{
                posX = self.size.width - ballHalfSize
                self.speedX *= -0.5
```

```
            }

            if posY < 0.0 + ballHalfSize {   ・・・I
                posY = 0.0 + ballHalfSize
                self.speedY *= -0.5
            } else if posY > self.size.height - ballHalfSize   {
                posY = self.size.height - ballHalfSize
                self.speedY *= -0.5
            }


            ball.position = CGPointMake(posX, posY)   ・・・J
        }

        // 加速度データを取得するハンドラをセット
        myMotionManager.startAccelerometerUpdatesToQueue(NSOperationQue
ue.currentQueue()!, withHandler: accelerometerHandler)   ・・・K
    }
}
```

＠と Aは、加速度センサーの値をボールのスピードとして利用するための変数宣言です。

Bは画面の中心位置を求める処理です。ここで求めた中心位置に Cで作成するボールを配置します。

Dは加速度センサーからのデータ取得間隔を1/60秒に設定しています。

Eの accelerometerHandlerは Kの設定によって呼び出されるようになります。

Fは加速度センサーの値を取得して変数speedXとsppedYに代入しています。

Gはボールの新しい位置を設定しています。現在位置に Fで求めた値を加算しています。

Hの ifブロックでは、ボールが画面の左端と右端に衝突した場合の処理です。衝突したかどうかは現在位置が「画面の左端の位置 + ボールの半分の幅」よりも左になったかどうか、または、「画面の右端の位置 - ボールの半分の幅」よりも右になったかどうかで判断しています。ボールの半分の幅を考慮しない場合は、ボールが左端(右端)からはみ出してしまうので注意していください。衝突した場合は、ボールの現在位置

を左端または右端に設定するとともに、反発したように見せるため「self.speedX *= -0.5」をしています。この-0.5を変更することで反発の度合いを変更することができます。
Iは Hと同様に、画面の上端と下端に衝突した場合の判定をしています。
最後に Jで、ボールの現在位置を設定しています。

15時間目 確認テスト

Q1 ここではUI部品を貼り付けて起動してみるだけなので、サンプルコードは掲載しません。
様々な部品を貼り付けて、表示を確認して見ましょう。

Q2 ここでは「15-3 キッチンタイマーの作成」を改造してタイマーをカウントアップする例を示します。
修正箇所は以下の通りです。

```
class InterfaceController: WKInterfaceController {

    @IBOutlet var sliderMinute: WKInterfaceSlider!
    @IBOutlet var timerMinute: WKInterfaceTimer!
    @IBOutlet var btnStart: WKInterfaceButton!

    var isStart = true
    var currentValue : Float = 5.0

    override func awakeWithContext(context: AnyObject?) {
        super.awakeWithContext(context)

        setTimer(0) //・・・@

    }

    // START/STOPボタン タップ時の処理
    @IBAction func btnStartPushed() { //・・・|
```

```
        if isStart {
            btnStart.setTitle("STOP")

            setTimer(0) //・・・A
            timerMinute.start()
        } else {
            btnStart.setTitle("START")
            timerMinute.stop()
        }

        isStart = !isStart

        sliderMinute.setEnabled(isStart)
    }

    // 省略
}
```

＠は画面表示時の値の設定です。カウントアップをするため0にしています。

Aはタイマーを停止させたときのタイマーの設定です。0にしてリセットを行っています。

setTimerメソッド内でtimerMinute.setDateの記述がありますが、「15-3 キッチンタイマーの作成」でも説明した通り、この引数にマイナスの値を与えない限りはカウントアップを行います。