Tylko głupiec może oczekiwać racjonalnego zachowania od innych ludzi.

Tao Programowania,
autor nieznany

Widoki i kontrolki

Widoki i kontrolki

- Przydatne **typy geometryczne**:
  - CGFloat - liczba zmiennoprzecinkowa (**float**)

```
typedef float CGFloat;
```

  - CGPoint - **struktura** reprezentująca punkt

```
struct CGPoint {
  CGFloat x;
  CGFloat y;
};

typedef struct CGPoint CGPoint;
```

Widoki i kontrolki

- Przydatne typy geometryczne:
  - **CGSize** - struktura, reprezentuje szerokość i wysokość
  - dostępne **makro**: **CGSizeMake**(CGFloat width, CGFloat height)

```
struct CGSize {
  CGFloat width;
  CGFloat height;
};

typedef struct CGSize CGSize;
```

  - **CGRect** - również **struktura**, która reprezentuje położenie i wielkość
  - makro: **CGRectMake**(CGFloat x, CGFloat y, CGFloat width, CGFloat height)

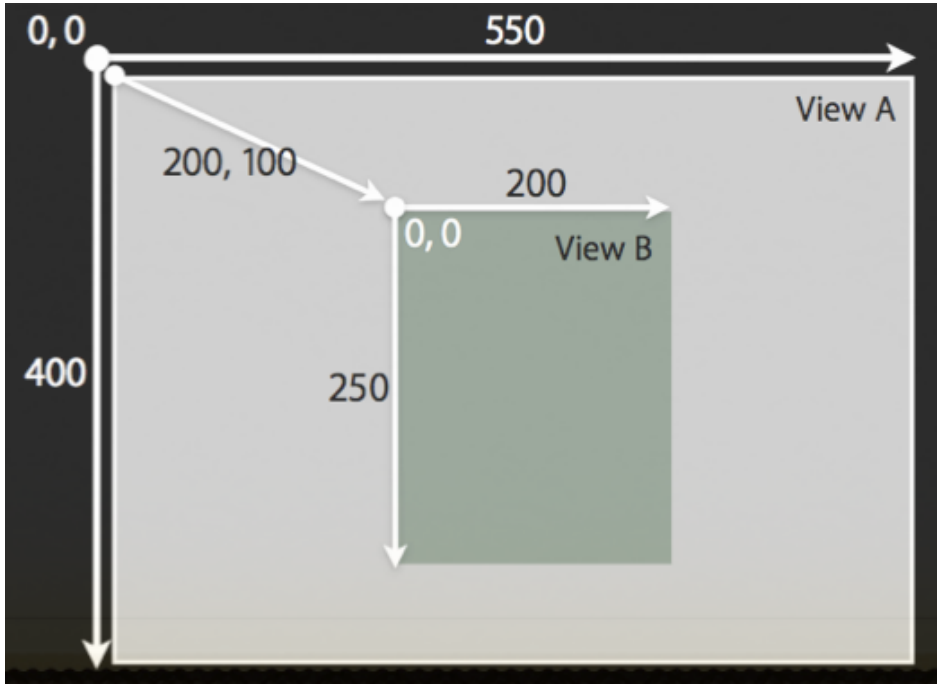
```
struct CGRect {
  CGPoint origin;
  CGSize size;
};

typedef struct CGRect CGRect;
```

Widoki i kontrolki

- FRAME VS BOUNDS
  - Położenie i wielkość można określić na 2 sposoby:
    - **Frame** - układ współrzędnych w relacji **widok-rodzic**
    - **Bounds** - lokalny układ współrzędnych (**x,y** zawsze jest równe **0,0**)

Widok **A Frame**:
origin: **0,0**
size: **550x400**

Widok **A Bounds**:
origin: **0,0**
size: **550x400**



Widok **B Frame:**
origin: **200,100**
size: **200x250**

Widok **B Bounds**:
origin: **0,0**
size: **200x250**

Widoki i kontrolki

- Klasa **UIScreen**:
  - dostarcza informacji na temat wielkości ekranu
  - szerokość: **320pt (640px** iPhone Retina Display)
  - wysokość: **480pt (960px** iPhone Retina Display) lub **568pt** (dla iPhone 5/5c/5s)

```
[[UIScreen mainScreen] bounds].size;
```

- Status bar (siec, zasięg, bateria) zajmuje 20pt

```
[UIApplication sharedApplication].statusBarHidden = YES;

CGRect frame = [[UIScreen mainScreen] applicationFrame];
```

- - Skala ekranu (1.0 dla non-retina oraz 2.0 dla wyświetlaczy retina)

```
[[UIScreen mainScreen] scale];
```

Widoki i kontrolki

- Hierarchia widoków
  - Jedno okno (**UIWindow**) + kilka widoków (**UIView**)
  - Dodając widok dodajemy go do okna lub widoku
  - każdy widok/kontrolka musi mieć widok nadrzędny (wyjątek: UIWindow)

- Struktura drzewiasta widoków
  - Okno jako korzeń drzewa
  - każdy widok ma dokładnie jednego rodzica

```
@property(nonatomic, readonly) UIView *superview;
```

- Każdy widok ma zero lub więcej podwidoków

```
@property(nonatomic, readonly, copy) NSArray *subviews;
```

Widoki i kontrolki

- Tworzenie widoków

```
CGRect frame = [UIScreen mainScreen].applicationFrame;
UIView *view = [[UIView alloc] initWithFrame:frame];
```

- Dodawanie widoku jako pod widoku (okna lub widoku)
  - superview wywołuje **retain** na dodanym widoku

```
- (void)addSubview:(UIView *)view;
```

- usuwanie z drzewa
  - wywołuje **release** na widoku

```
- (void)removeFromSuperview;
```

Widoki i kontrolki

- Rysowanie 2D
  - Widok przygotuje środowisko, aby ułatwić rysowanie
  - Korzystanie z funkcji Quartz 2D do rys. kształtów
  - **UIGraphicsGetCurrentContext()**;
  - CoreGraphics/CGContext.h
  - Korzystanie z funkcji NSString, UIImage...
  - drawAtPoint, drawInRect

```
- (void)drawRect:(CGRect)rect;
```

- **Uwaga!**
  - Nigdy nie należy wywoływać metody **drawRect** wprost!
  - Korzystamy z [widok **setNeedsDisplay**];
  - system wybierze najlepszy czas na odświeżenie widoku

Widoki i kontrolki

- **UIColor**
  - Paleta barw podstawowych, m.in:

```
UIColor *redColor = [UIColor blueColor]
```

  - możliwość definiowania nowych:

```
+ (UIColor *)colorWithWhite:(CGFloat)white alpha:(CGFloat)alpha;
+ (UIColor *)colorWithRed:(CGFloat)red
    green:(CGFloat)green
    blue:(CGFloat)blue
    alpha:(CGFloat)alpha;

+ (UIColor *)colorWithCGColor:(CGColorRef)cgColor;
+ (UIColor *)colorWithPatternImage:(UIImage *)image;
```

Widoki i kontrolki

- UIFont
  - dostęp do czcionek systemowych
  - dostęp do czcionek ‘po nazwie’
  - od iOS 4 - możliwość dodawania własnych fontów (wymagana edycja pliku info.plist)

```
[UIFont systemFontOfSize:14.0];
[UIFont fontWithName:@"Courier-Bold" size:14.0];
```

- niestety, są również spore ograniczenia
- brak możliwości zdefiniownia odstępu między znakami
- brak możliwości zdefiniownia odstępu między wierszami
- różne zestawy fontów na urządzeniu a na symulatorze
- różne zestawy fontów między różnymi wersjami SDK

```
NSMutableArray *fonts = [NSMutableArray arrayWithArray:[UIFont familyNames]];
[fonts sortUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
NSLog(@"fonts = %@", fonts);
```

Widoki i kontrolki

- metody do obsługi prostych gestów:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
```

- Jednak aktualnie zalecane jest stosowanie klas na bazie klasy UIGestureRecognizer i jego subklas:
  - UITapGestureRecognizer
  - UIPanGestureRecognizer
  - UISwipeGestureRecognizer
  - UIPinchGestureRecognizer
  - UIRotationGestureRecognizer
  - UILongPressGestureRecognizer

Widoki i kontrolki

- Proste Animacje:
  - dowolne atrybuty widoku mogą w prosty sposób być animowane
  - klasa UIView dostarcza metod pozwalających na:
  - przesuwanie
  - zmianę rozmiaru

- zmiany atrybutów i szczegóły animacji między (starsze podeście)
- beginAnimations: context:
- commitAnimations:

- animowanie za pomocą bloków

```
+ (void)animateWithDuration:(NSTimeInterval)duration
  animations:(void (^)(void))animations
  completion:(void (^)(BOOL finished))completion;
```

Widoki i kontrolki

- Graficzne obiekty, z którymi użytkownik może działać
- pola tekstowe
- przyciski
- przełączniki
- suwaki
- i inne

- dziedziczą po klasie UIControl
- NSObject <- UIResponder <- UIView <- UIControl

- typowe zachowania kontrolek obsługiwane są przez klase UIControl

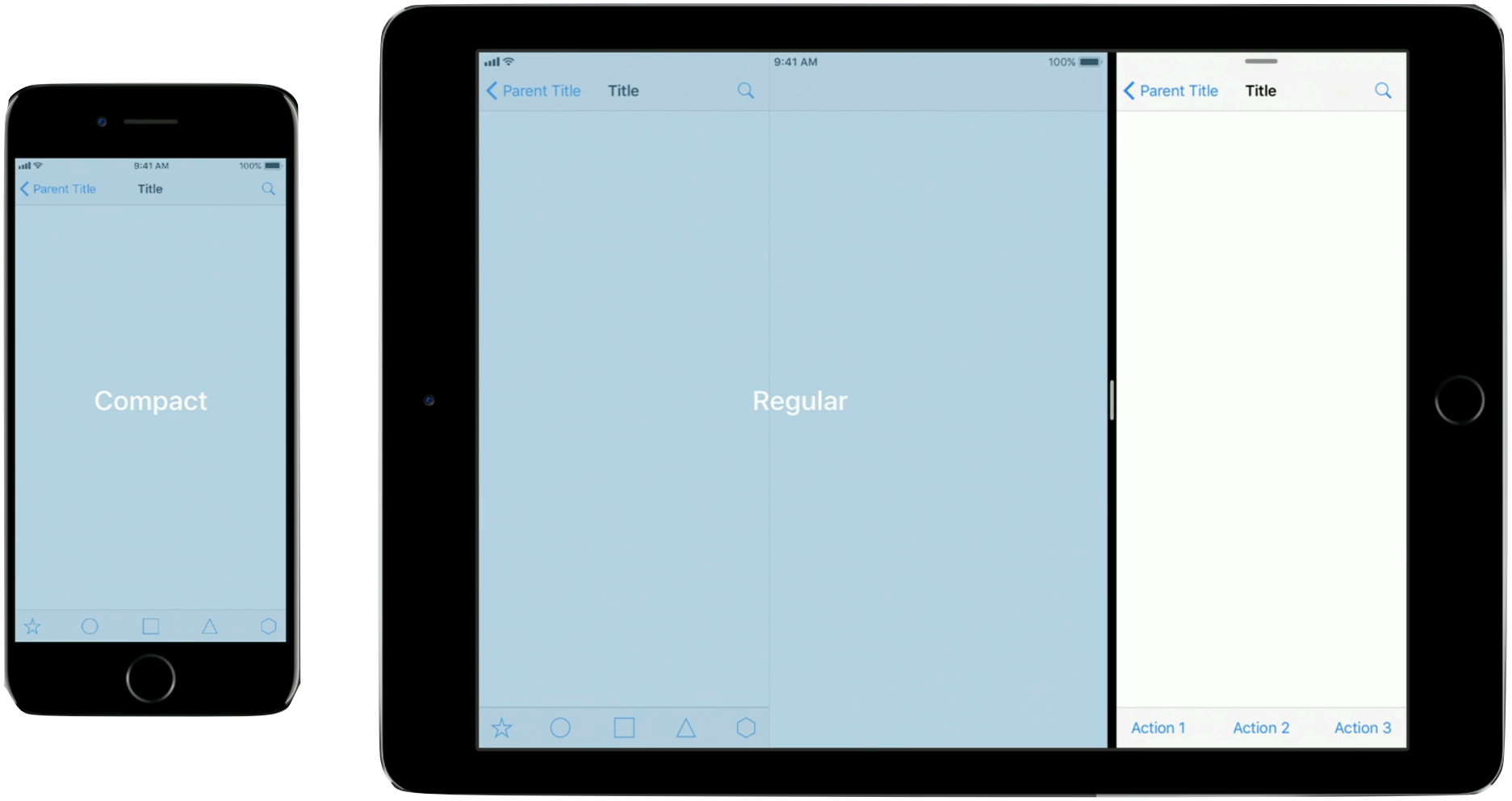
Widoki i kontrolki

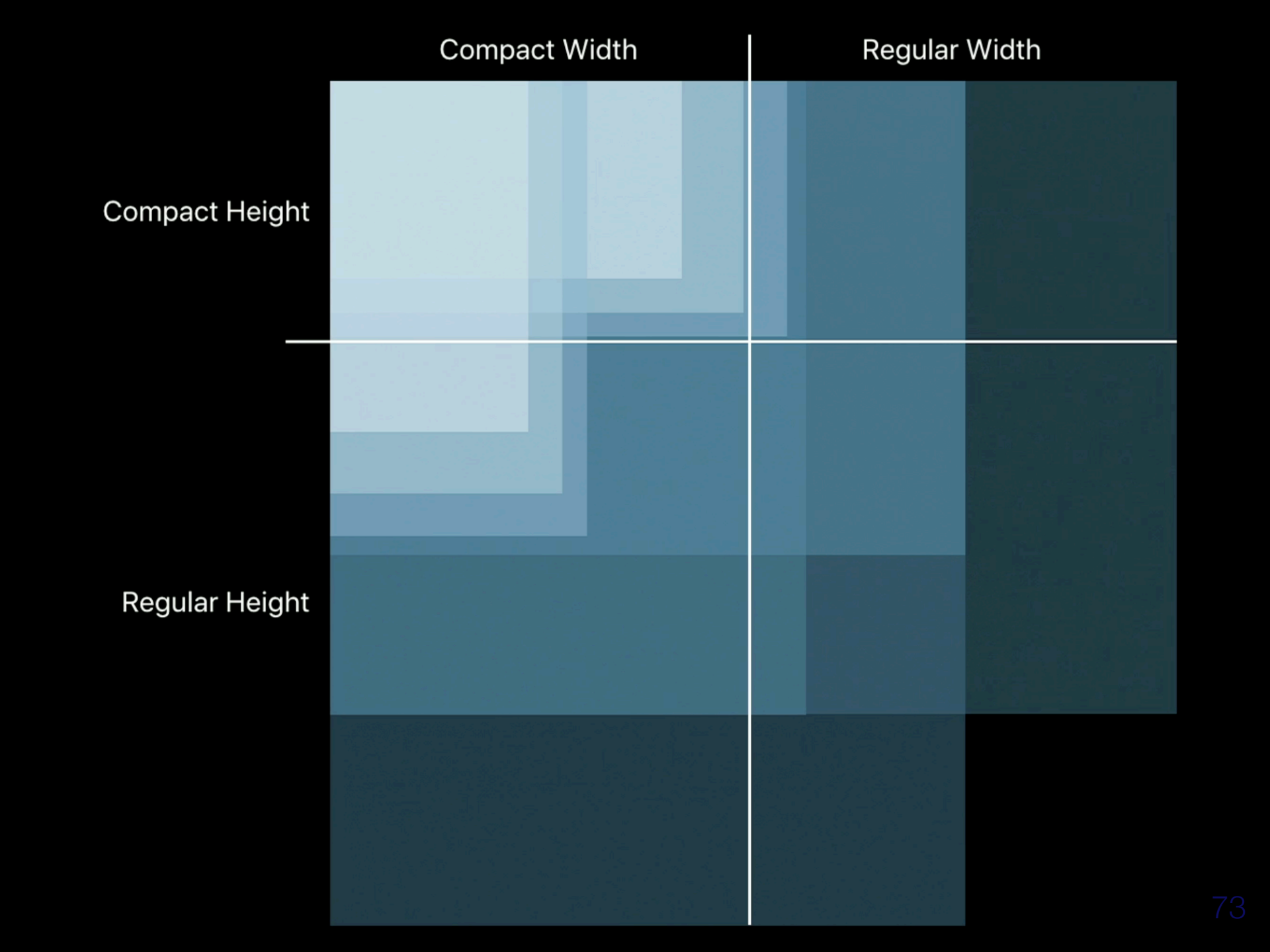
- Popularne atrybuty kontrolek:
  - enabled
  - highlighted
  - selected
  - state

```
@property(nonatomic, getter=isEnabled) BOOL enabled;
@property(nonatomic, getter=isHighlighted) BOOL highlighted;
@property(nonatomic, getter=isSelected) BOOL selected;
@property(nonatomic, readonly) UIControlState state;
```

- Możliwe stany kontroli to:
  - UIControlStateNormal
  - UIControlStateHighlighted
  - UIControlStateDisabled
  - UIControlStateSelected

Size Classes




Compact Width
Regular Width
Compact Height
Regular Height

Zastosowania oraz implementacja UITableView

UITableView

Przeznaczenie kontrolki:

- wyświetlanie dużych ilości danych w kolumnie
- zoptymalizowana na urządzenia mobilne
- recykling komórek
- stosowana również do nawigacji
- może być stosowana w dowolnej instancji klasy UIViewController,

bądź jako samodzielna kontroler jako UITableViewController

UITableView

- UITableView jest tylko odpowiedzialny za wyświetlanie danych a nie za ich przetrzymywanie.

- Developer sam decyduje o najlepszym sposobie przechowywania danych, np.
  - w pamięci
  - w zapisanych plikach
  - w bazie danych

- UITableView musimy obsłużyć protokół UITableViewDataSource.
- Dwie metody są wymagane:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section;
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath;
```

UITableView

Opcjonalny protokół to UITableViewDelegate.

- wszystkie metody są opcjonalne

Ten protokół umożliwa nam modyfikację zachowania i wyglądu, np.

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath;
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath;
```

Zdeklarowanie się do obsługi protokołów:

```
//
//  SampleViewController.h
//  AppName
//

#import <UIKit/UIKit.h>

@interface SampleViewController : UIViewController <UITableViewDelegate, UITableViewDataSource>

@end
```

UITableView

- UITableView jest podzielony na sekcje.

- Każda sekcja może mieć własny header oraz footer view oraz określoną liczbę rows, czyli liczbę poszczególnych komórek.
- Przykłady implementacji:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 10;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {

    int sections = [[self myArray] count];

    return (sections == 0) ? 1 : sections;
}
```

UITableView

- Liczba komórek w sekcji:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    int rows;

    switch (section) {
        case 2:  rows = 5; break;
        case 4:  rows = 6; break;
        default: rows = 3; break;
    }

    return rows;
}
```

- Musimy rozpoznać sprawdzić o którą sekcję UITableView nas pyta (zmienna section)
  - najczęściej do tego celu stosuje się instrukcję switch
- Przy większej ilości sekcji warto stosować zdefiniowane stałe zamiast liczb.

UITableView

- Liczba komórek w sekcji, przykład 2:

```
#define kSectionCities 1
#define kSectionPeople 2
#define kSectionPlaces 4

- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {


    int rows;

    switch (section) {
        case kSectionCities:  [[self CitiesArray] count]; break;
        case kSectionPeople:  [[self PeopleArray] count]; break;
        case kSectionPlaces:  [[self PlacesArray] count]; break;
        default: rows = 0; break;
    }

    return rows;
}
```

UITableView

- Liczba komórek w sekcji, przykład 3:

```
typedef NS_ENUM(NSUInteger, kMyTableSessionType) {
    kMyTableSessionTypeCities = 0,
    kMyTableSessionTypePeople = 1,
    kMyTableSessionTypePlaces = 4
};

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section {

    int rows;

    switch ((kMyTableSessionType)section) {
        case kMyTableSessionTypeCities:  [[self CitiesArray] count]; break;
        case kMyTableSessionTypePeople:  [[self PeopleArray] count]; break;
        case kMyTableSessionTypePlaces:  [[self PlacesArray] count]; break;
        default: rows = 0; break;
    }

    return rows;
}
```

UITableViewCell

- NSIndexPath jest używany do opisania położenia komórki i posiada dwie wartości tylko do odczytu: section oraz row (dla collectionViews można stosować section / item)

```
int section = [indexPath section];
int row = [indexPath row];
```

- Poniższa metoda jest wywoływana dla każdej widocznej komórki w tabeli.
- Tylko za pomocą NSIndexPath jesteśmy w stanie określić jakimi danymi uzupełnić wyświetlaną komórkę.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = …

    // Configure the cell.

    return cell;
}
```

UITableView - przykład tworzenia komórki

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    int section = [indexPath section];
    int row = [indexPath row];

    static NSString *CellIdentifier = @“Cell";

    UITableViewCell *cell = nil;
    cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:
                        UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
    }

    // Configure the cell.

    return cell;
}
```

UITableView - przykład tworzenia komórki

- Od wersji iOS 6 dostępna jest dodatkowa metoda

```
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
                                                        forIndexPath:indexPath];
```

- gwarantuje ona, że zwróci ona już zaalokowaną komórkę
- aby to rozwiązanie działało prawidłowo należy zarejestrować odpowiednią klasę (lub plik xib) dla odpowiedniego identyfikatora - jest to metoda wywoływana automatycznie gdy użyte są pliki storyboard

```
[self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier:@"Cell"];

[self.tableView registerNib:[UINib nibWithNibName:@"CustomerCell" bundle:nil]
     forCellReuseIdentifier:@"CustomerCell"];
```