Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Basics of Mobile Application Development

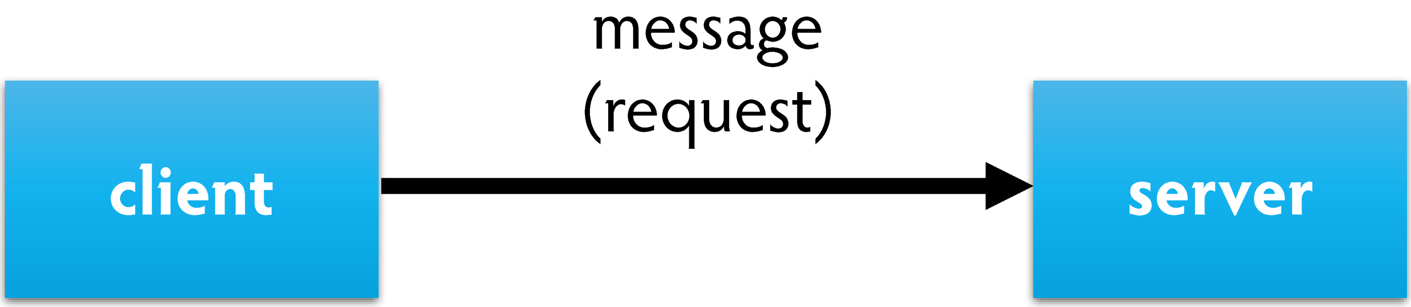
Objective-C

Reminder

OOP – keywords

- Object
  - Represents of entities of the real world
- Class of objects
  - Group of similar objects
    - Behavior
    - Structure
  - Template to create objects
- Method
  - A function (procedure) which manipulates the state of an object
- Field
  - A variable defining a property of an object
- Messaging
  - Interaction of objects
  - Interfaces are defined to facilitate the communication of objects
- Abstraction
  - Grouping classes
- Hierarchy
  - Design and implementation tool

Client sends a message



Object,
requires a
service

Calling a public
method

Object,
executes tasks
upon request

Objective-C

Basic properties

- Extension of C language
  - This it not C++
  - Thin layer on the C, which is processed by the preprocessor
  - New syntactic elements to create classes and methods
    - Smalltalk-style
  - Fully object oriented
    - The C variables, functions are the same
      - All C code can be compiled with the Objective-C compiler
  - The iOS framework was Objective-C based originally
    - The are existing Objective-C based codes, libraries
- Short history
  - Obective-C was developed by the beginning of 1980s
  - NeXT bought the license
  - Apple acquired Next

Data types

- C primitives
  - Without explicit type: `void`
  - Integers: (`unsigned`) `short`, `char`, `int`, `long`, `long long`
  - Fix size integers: `int8_t`, `uint16_t`, …
  - Floating-point number: `float`, `double`, `long double`
- Objective-C primitives
  - Logical: `BOOL` (two values: `YES` and `NO`)
  - Base type of the objects: `id`
  - Data type for instances of meta-classes: `Class`
  - Type for storing selectors (to store functions): `SEL`
- Types can be used as usual, examples:

  -
    ```
    short aShort = 1234;
    char aChar = 'A';
    BOOL isGood = YES;
    ```

Data types

- Basic objects
  - Objects: `NSObject`
    - Superclass most of the objects
  - Numbers: `NSNumber`
    - Immutable – the same reasons as in Java
  - Text: `NSString`
    - Immutable
  - Fixedpoint numbers: `NSDecimalNumber`
    - Immutable
  - Collections:
    - Set (immutable): `NSSet`
    - Array (immutable): `NSArray`
    - Key-Value pairs (immutable): `NSDictionary`
  - Date: `NSDate`
- Each above can be accessed through pointers
  - The instantiation and lifetime will be discussed

Classes

- The concept of the classes and objects are the same
  - However some keywords are used in other way
- Class declaration is separated into:
  - .h file – header, which contains the public interface of class/object
  - .m file – implementation, which contains the private implementation
- On the following slides the .h file is on the left side and the .m file is on the right side
- Remember! New keywords starts with the @ symbol, as the preprocessor has to find them

Creating class

Card.H file

- A `Card` class will be created, which is subclass of the `NSObject`.
- The superclass must be imported.

```
#import <Foundation/Foundation.h>

@interface Card : NSObject
// Public declarations

@end
```

Card.M file

- Implementation of the same class. (You do not have to specify the superclass again)
- The .h file must be imported.

```
#import "Card.h"

@implementation Card
// Implementation

@end
```

.h import

- Previously an element of the framework has been imported
- To import the entire framework
  - `@import Foundation;`
- To import anything else
  - `#import "Superclass"`

@interface and @end

- Between the two keywords you can specify the interface if the class
  - The fields and the methods can be specified
    - In the .h file the public, int the .m file the private members
    - There is no other visibility level
- In case of .m file
- #import "Card.h"

```
#import "Card.h"

@interface Card()
// Private declarations
@end


@implementation Card


@end
```

`@interface` and `@end`

- Fields can be part of classes, which can be considered as the properties of the class/object
  - You can declare by using the `@property` keyword
    - The type and variable name must be specified
  - The declarations of get and set are also there
  - Members can only be accessed through methods
    - Public and private environment as well
  - Declaration is in the interface
  - Example
    - `@property (strong) NSString *contents;`
  - In this example a pointer refers to an `NSString`
    - If the object is a property, is must be access by using pointers
    - This brings the problem of memory management

@property

- A property can be `strong` or `weak`
  - `strong`: The object that is referred by the property, exists while at least one strong pointer refers to that specific object. (The number of reference is greater than zero.)
    - If you set it to `nil` the number of reference is decreased.
  - `weak`: If there is no strong pointer to that instance, then the object can be destroyed and the memory can be freed up.
    - The weak pointer is set to `nil` in that case.
- A property can be `nonatomic` as well
  - Then the access is not thread safe
  - In the other case, the compiler creates locks, and through of them the parallel access is controlled
  - Currently you can used `nonatomic`

@synthesize

- Behind the property there is a variable, which is declared by the compiler, along with the get and set functions
  - Its name is the name of the property, with an _ before the name
  - You can override this behavior by using the `@synthesize` keyword
  - Previous example can be continued: In the `@implementation` part of the .h file:
    - `@synthesize contents = _contentsvariable;`
- Of course, you can write your own get and set messages
- The code that is created automatically is something like this:
  - `@synthesize contents = _contents`

  -
    ```
    - (NSString *)contents
    {
        return _contents;
    }
    ```

  -
    ```
    - (void)setContents:(NSString *)contents
    {
        _contents = contents;
    }
    ```

Further options

- A property can not only be an object
  - `@property (nonatomic) BOOL chosen;`
  - `@property (nonatomic) BOOL matched;`
  - Here there is no meaning to use strong/weak options, as they are not stored in the heap of the memory.
  - Thus they exists till the object exists
  - Arbitrary C type can be used, even `structs`
- You can specify the name of the get/set message
- Previous example
    - `-(BOOL)chosen`
  - Instead of the previous:
    - `@poperty (nonatomic, getter=isChosen) BOOL chosen;`
      `@poperty (nonatomic, getter=isMatched) BOOL matched;`
  - Then
    - `-(BOOL)isChosen`
  - The readability of the code is better
- A property also can be `readonly` as well
  - And several others, which are not important at this point

Functions – Messages

- Instead of calling methods/functions, the message sending semantics comes into foreground
  - C++ style approach (traditional)
    - `foo->bar (parameter);`
  - Objective-C approach
    - `[foo bar:parameter];`
- It is determined in runtime whether the target object can or cannot process the request
  - Thus the type checking happens in runtime not in compilation time
  - Allways expect NIL as response
- Additional information
  - The different parameters are defined through the name of the message
  - Traditionally
  - `-(type)method:(type)param1 :(type)param2;`
  - Objective-C
    `-(type)method:(type)param1 andParam2:(type)param2;`

Overload!?

- We would like to have two messages with different parameter type
  - `-(int)doIt:(int)param1 :(int)param2;`
  - `-(int)doIt:(int)param1 :(NSString*)param2;`
- This is not allowed
- But if you include the name (purpose) of the parameter into the name of the message
  - `-(int)doIt:(int)param1 withSomeInt:(int)param2;`
  - `-(int)doIt:(int)param1 withSomeString:(NSString*)param2;`
- In that case the two messages have different names, so it is not overloading
  - Overload is not supported by Objective-C
  - But you can mimic, by using the id type
    - And the implementation decides what to do with the actual parameter
- In previous case, the two messages both have two parameters
  - Neither one is optional

Card example

Card.H file

```
#import <Foundation/Foundation.h>

@interface Card : NSObject


@property (strong) NSString
*contents;


@property (nonatomic,
getter=isChosen) BOOL chosen;

@property (nonatomic,
getter=isMatched) BOOL matched;

-(int)match:(Card *)card;



@end
```

Card.M file

- `#import "Card.h"`

```
@interface Card()
// Private declarations
@end

@implementation Card

-(int)match:(Card *)card
{
   int score = 0;

   // We calculate the score

   return score;
}

@end
```

New message

-
```
-(int)match:(Card *)card
{
    int score = 0;

    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }

    return score;
}
```

- Observe
  - You send the message as previously mentioned
  - Instead of `this` there is `self`
  - As everything is an object, you can use . to access members
  - The name of the `isEqualToString`
  - The `self.contents` is the get message, similarly to `card.contents`

Comparison

- The == operator compares the value in case of primitives and objects (pointers) as well
  - Unsurprisingly, the memory addresses are compared
- In case of objects you must specify a message, which can compare the objects based on their properties
- `NSString:`
  - `isEqualToString`
- `NSNumber:`
  - `isEqualToNumber`
- Etc.

Extend the message – NSArray

- The signature of the message is extended
  - `-(int)match:(NSArray *)othercards;`
- Then the implementation will be

  -
    ```
    -(int)match:(NSArray *)othercards
    {
        int score = 0;

        for (Card *card in othercards) {
            if ([card.contents isEqualToString:self.contents]) {
                score = 1;
            }
        }

        return score;
    }
    ```
- You can observe the for-each loop
  - The syntax of the for loop is the well known one

Using the class

Deck.H

-
  ```
  #import <Foundation/Foundation.h>
  #import "Card.h"

  @interface Deck : NSObject

  -(void)addCard:(Card *)card
  atTop(BOOL)atTop;

  -(void)addCard:(Card *)card

  -(Card *)drawRandomCard;

  @end
  ```

Deck.M

-
  ```
  #import "Deck.h"

  @interface Deck()
  @end

  @implementation Deck

  -(void)addCard:(Card *)card
  atTop(BOOL)atTop
  {
     // TODO
  }

  -(void)addCard:(Card *)card
  {
     [self addCard:card atTop:NO];
  }

  -(Card *)drawRandomCard
  {
     // TODO
  }

  @end
  ```

In previous code

- There are two versions of `addCard` message
  - If you want to delegate, then you can send the other message
- There is no data structure to store the card data
  - And there is no code to manage the data
- `NSArray`
  - It is immutable, thus it is not feasible
  - However there is the `NSMutableArray` type
  - We will send messages to the array
    - Indexing => sending a message

Using the code

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards;
@end

@implementation Deck

-(void)addCard:(Card *)card atTop(BOOL)atTop
{
   if (aTop) {
      [self.cards insertObject:card atIndex:0];
   } else {
      [self.cards addObject:cards];
   }
}

…

@end
```

Using the code

- `@property (strong, nonatomic) NSMutableArray *cards;`

- This line creates the property and the variable of the property
- However the object is not initialized, so we have to it, and also, we have to manage the variable
  - Currently we have a problem, as in `addCard` we access to a `NIL` pointer
- The automatically generated get function:
  - `-(NSMutableArray *)cards { return _cards; }`
- It has to be replaced:

  -
    ```
    -(NSMutableArray *) cards {
       if (!_cards) _cards = [[NSMutableArray alloc] init];
       return _cards;
    }
    ```
- And we are now arriving to the important question of initialization

Initialization of an object

- The memory for the object (of a pointer) has to be allocated and the object also has to be initialized
  - Previously, we used the new operator and we called a constructor
  - In Objective-C there are no such things, then we must send two different messages
  - Technically the two messages can be separated, but we should not do that
    - Also it is not forbidden to return `NIL` after initialization
- Initialization with literals
  - `NSString: @"Hi guys";`
  - `NSNumber: @42;`
  - `NSArray: @[@"One", @"Two", @"Three"];`
  - You do not have to deal with the lifecycle
- Initialization of an object
  - `[[NSMutableArray alloc] init]`

drawRandomCard

```
-(Card *)drawRandomCard
{
   Card *randomCard = nil;
   if ([self.cards count]) {
      unsigned index = arc4random() % [self.cards count];
      randomCard = self.cards[index];
      [self.cards removeObjectAtIndex:index];
   }
   return randomCard;
}
```

- `cards[index]` is a message as well!

Create a subclass

PlayingCard.h

-
  ```
  #import "Card.h"

  @interface PlayingCard : Card

  @property (strong, nonatomic)
  NSString *suit;

  @property (nonatomic) NSUInteger
  rank;

  @end
  ```

PlayingCard.m

-
  ```
  #import "PlayingCard.h"
  @implementation PlayingCard

  - (NSString *)contents
  {
    return
    [NSString stringWithFormat:@"%d%@",
         self.rank, self.suit];
  }

  @end
  ```
- The get function of the contents is overriden here
- The variable is declared only once, in the superclass
- The string is not allocated, but cretaed by formatting

Continue

-
  ```
  #import "PlayingCard.h"
  @implementation PlayingCard

  - (NSString *)contents
  {
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
  }

  @end
  ```
- Here the [ ] and @[ ] statements are translated to sending messages
- We can create our own get and set messages

-
  ```
  - (NSString *)suit
  {
     return _suit ? _suit : @"?";
  }
  ```

Lets continue

- Creating a set message to set the suit of the cards
  - However we face a problem, as neither the get and set message is generated automatically
  - Thus the property variable will not be synthesized
  - We have to do it manually
  - The property variable must not be accessed directly outside of the set/get messages
- `@synthesize suit = _suit;`
- The code
  - Remember, you can send a message to any object
-
```
- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}
```

Class members

- Previous members was instance members
  - To create class members you have to used the + symbol

  -
    ```
    + (NSArray *)validSuits
    {
        return @[@"♥",@"♦",@"♠",@"♣"];
    }
    ```
- In this case you have to send the message to the class

  -
    ```
    - (void)setSuit:(NSString *)suit
    {
        if ([[PlayingCard validSuits]
    containsObject:suit]) {
              _suit = suit;
        }
    }
    ```
- You can create public and private class members as well

Current state

PlayingCard.h

-
  ```
  #import "Card.h"

  @interface PlayingCard : Card

  @property (strong, nonatomic) NSString *suit;

  @property (nonatomic) NSUInteger rank;

  + (NSArray *)validSuits;
  + (NSArray *)rankStrings;
  + (NSUInteger)maxRank;

  @end
  ```

PlayingCard.m

-
  ```
  #import "PlayingCard.h"
  @implementation PlayingCard
  @synthesize suit = _suit;
  - (NSString *)contents
  {
   NSArray *rankStrings = [PlayingCard rankStrings];
   return [rankStrings[self.rank]
                   stringByAppendingString:self.suit];
  }
  + (NSArray *)validSuits
  {
     return @[@"♥",@"♦",@"♠",@"♣"];
  }
  + (NSArray *)rankStrings
  {
     return @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
  }
  - (void)setSuit:(NSString *)suit
  {
     if ([[PlayingCard validSuits] containsObject:suit]) {
        _suit = suit;
     }
  }
  -(NSString *)suit
  {
     return _suit ? _suit : @"?";
  }
  + (NSUInteger)maxRank { return [[self rankStrings] count]-1; }
  @end
  ```

`PlayingCardDeck`

- This will contain the PlayingCards
  - Based on the Deck class
  - There is no extension in the interface part
  - Existing cards will be inserted during the initialization
- `init`
  - Unusual – compared to the well-known constructors
  - There is a return type – the type of the instance (`instancetype`)
  - The created instance is assigned to the `self` variable
  - The `init`, or any alternative have to be called immediately after the `alloc` call
    - Even if they can be separated technically

New class

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"

@implementation PlayingCardDeck

- (instancetype)init
{
   self = [super init];
   if (self) {


   }
   return self;
}

@end
```

- Note
  - There is a return!
  - Superclass is initialized first
  - It can be resulted in `NIL`

PlayingCardDeck.m

```
#import "PlayingCardDeck.h„
#import "PlayingCard.h"

@implementation PlayingCardDeck

- (instancetype)init
{
   self = [super init];
   if (self) {
      for (NSString *suit in [PlayingCard validSuits]) {
         for (NSUInteger rank = 1; rank <= [PlayingCard maxRank]; rank++) {
            PlayingCard *card = [[PlayingCard alloc] init];
            card.rank = rank;
            card.suit = suit;
            [self addCard:card];
         }
      }
   }
   return self;
}

@end
```

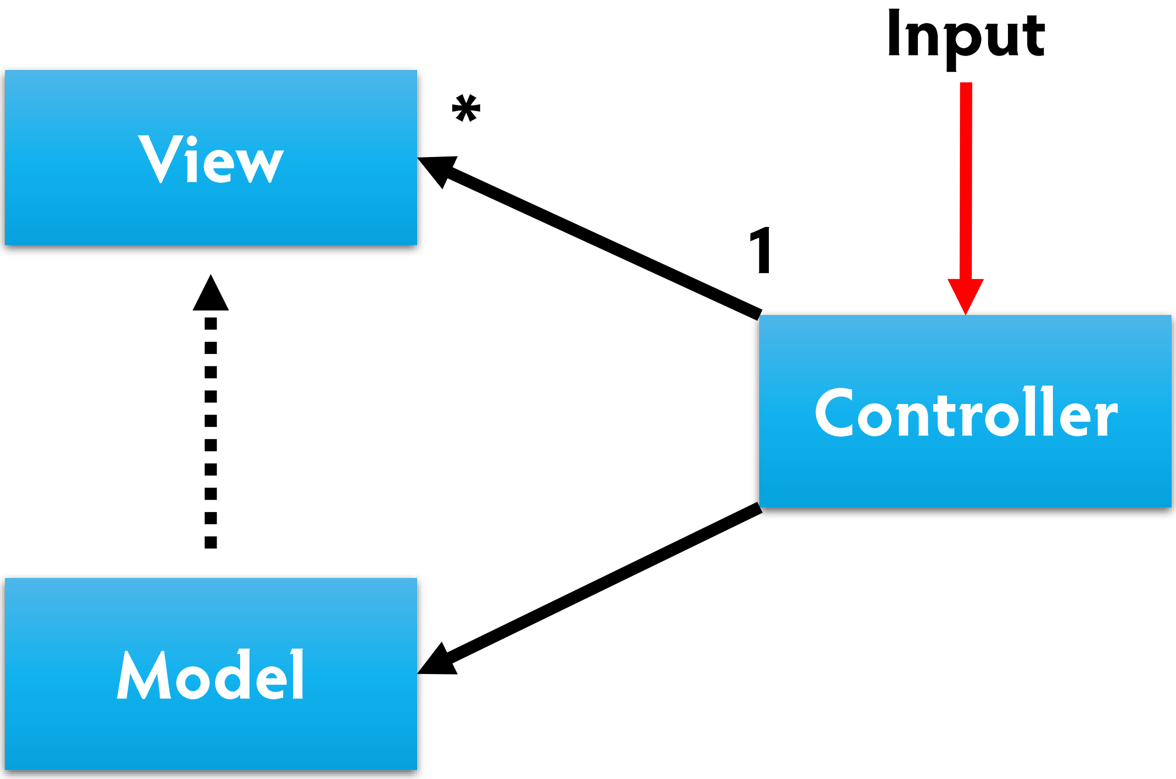
Question

- What does the next line do?

```
cardA.contents = @[cardB.contents,cardC.contents][[cardB match:@[cardC]] ? 1 : 0]
```

MVC

MVC



MVC

- The application has three layers
  - Model
    - The representation of the information stored by the application
      - Plain data is augmented with meta data to provide meaning
    - Many application uses permanent storing procedure to save data
    - The data access layer is part of the model, most of the cases

MVC

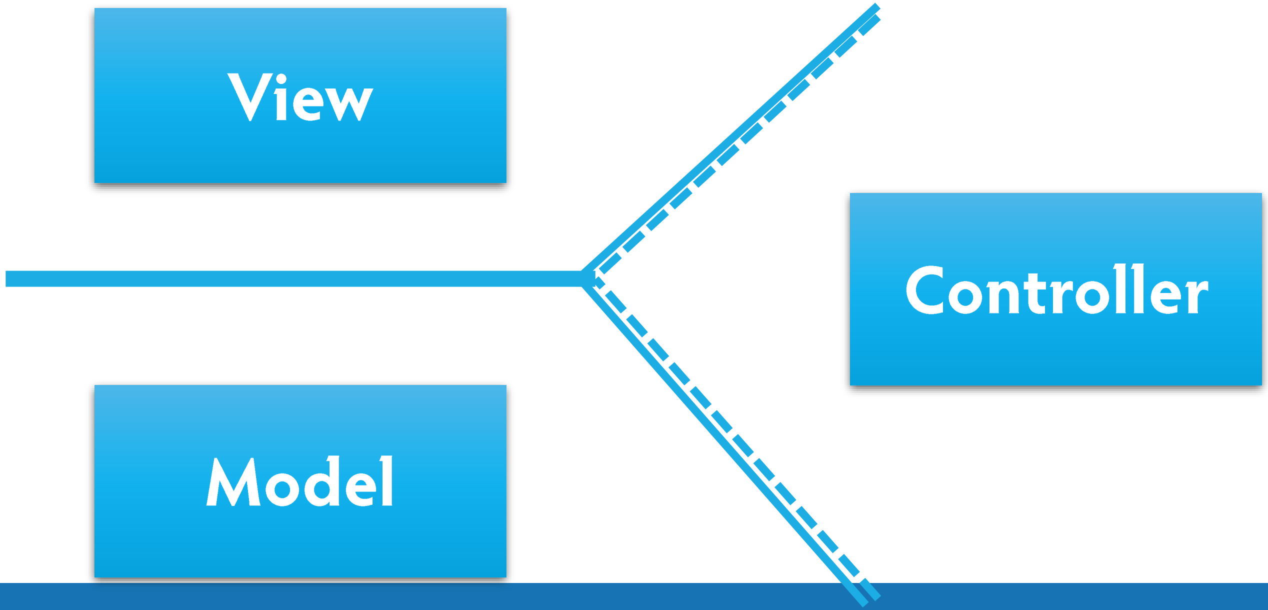
- The application has three layers
  - View
    - Visualize the model in the correct form, which is capable of user interaction
    - Typically it is a UI element
    - Different view for different objective may be exist

MVC

- The application has three layers
  - Controller
    - Events (mostly user interactions) are processed and appropriate response is generated
    - May change the model

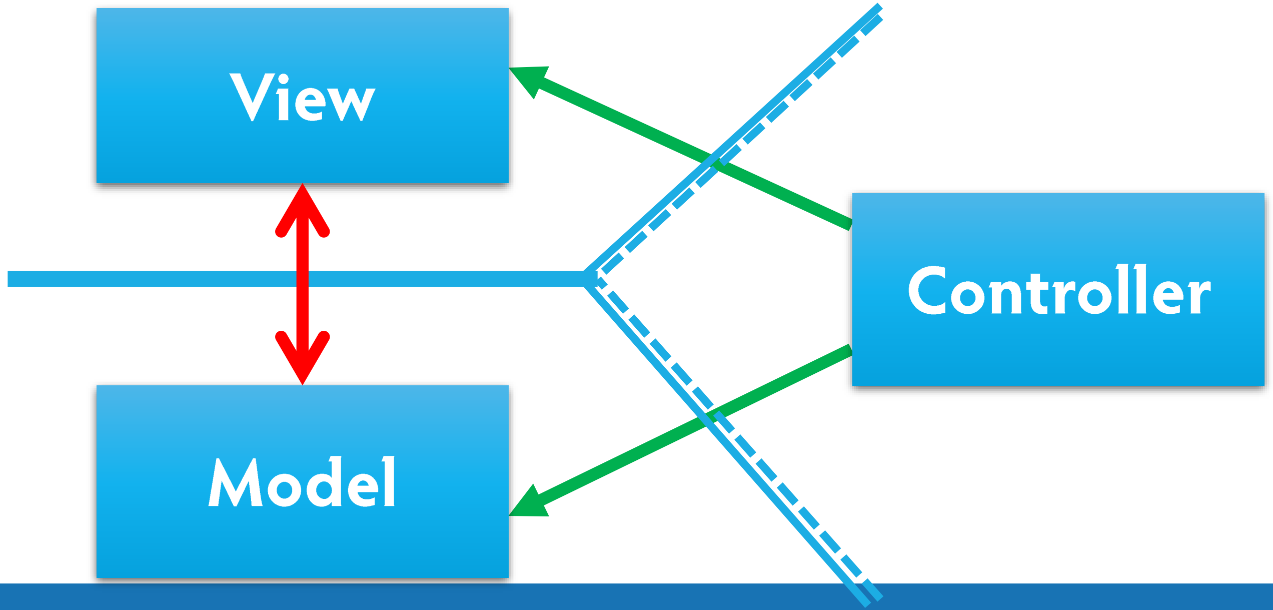
MVC

- Communication between the components



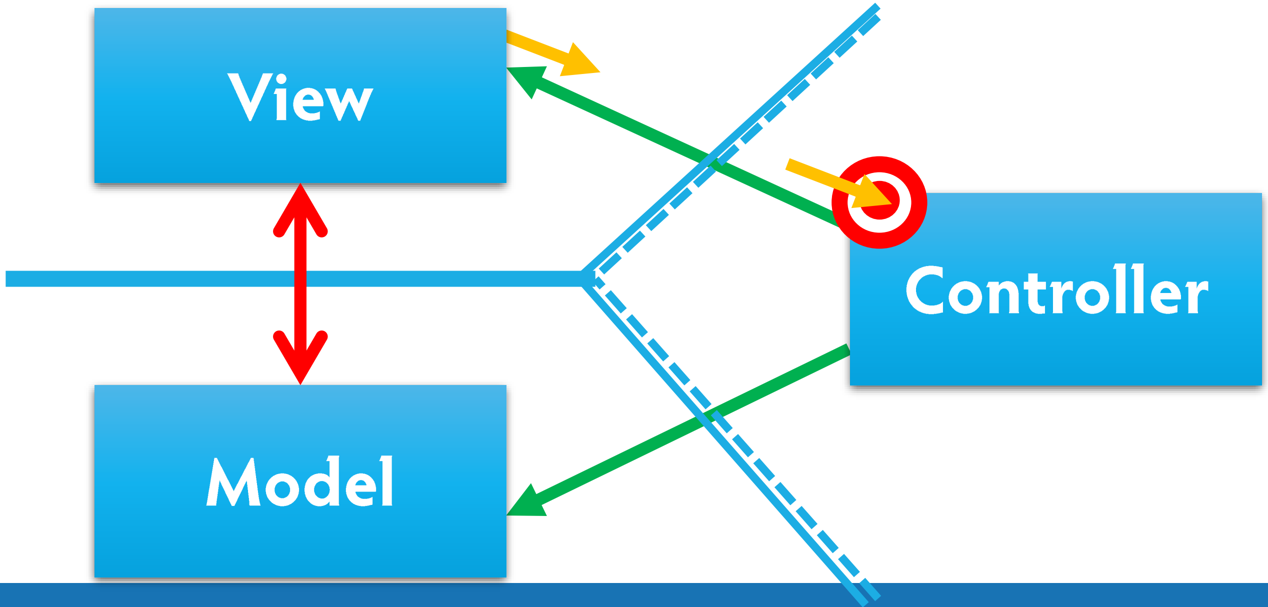
MVC

- Communication between the components
  - Controller communications with View and Model
  - View and Model cannot access each other directly



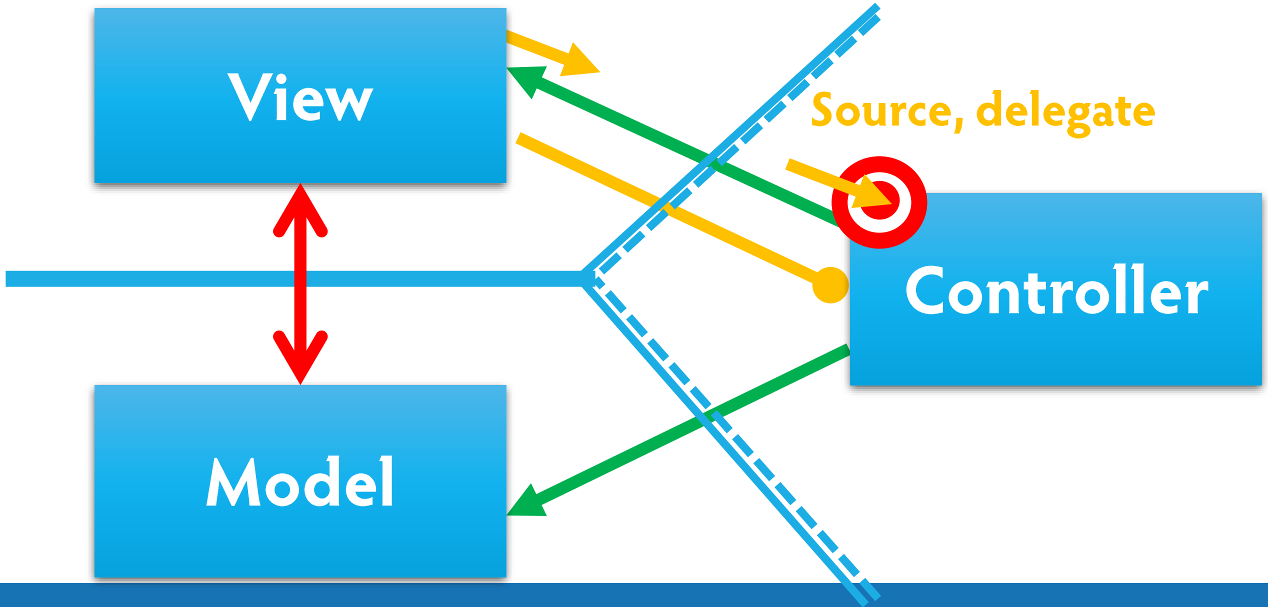
MVC

- However View notifies the controller indirectly
  - Controllers have to specify Targets
  - You can invoke Actions for specific Targets



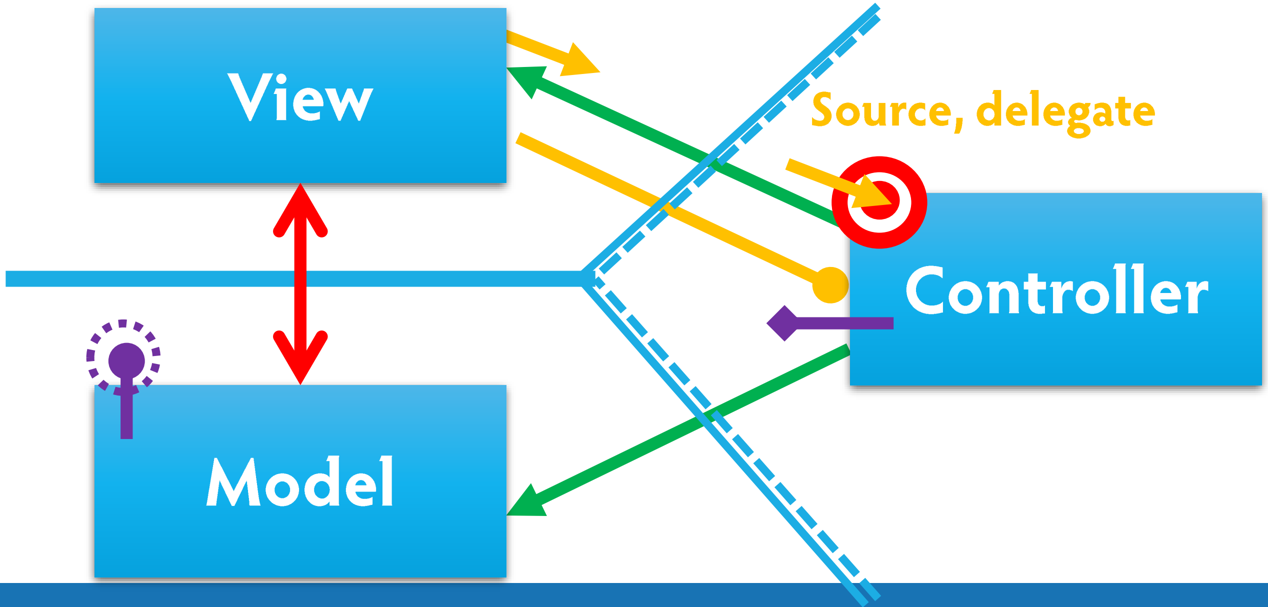
MVC

- In other cases the View has to be synchronized
  - Delegates have to be created
  - View have to know the structure of the data
  - Controller has to be act as a source of data
    - Data is transformed by the Controller

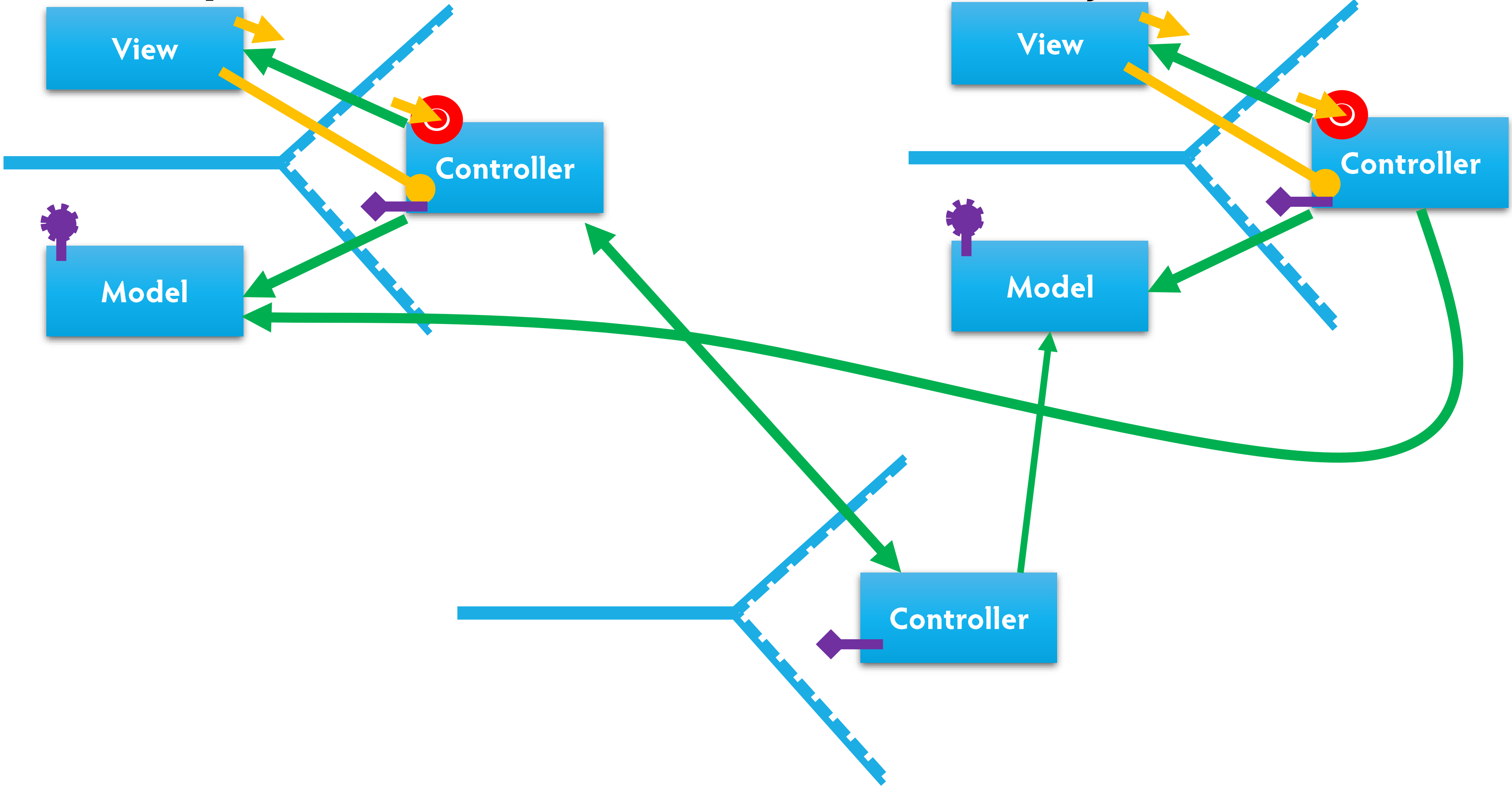


MVC

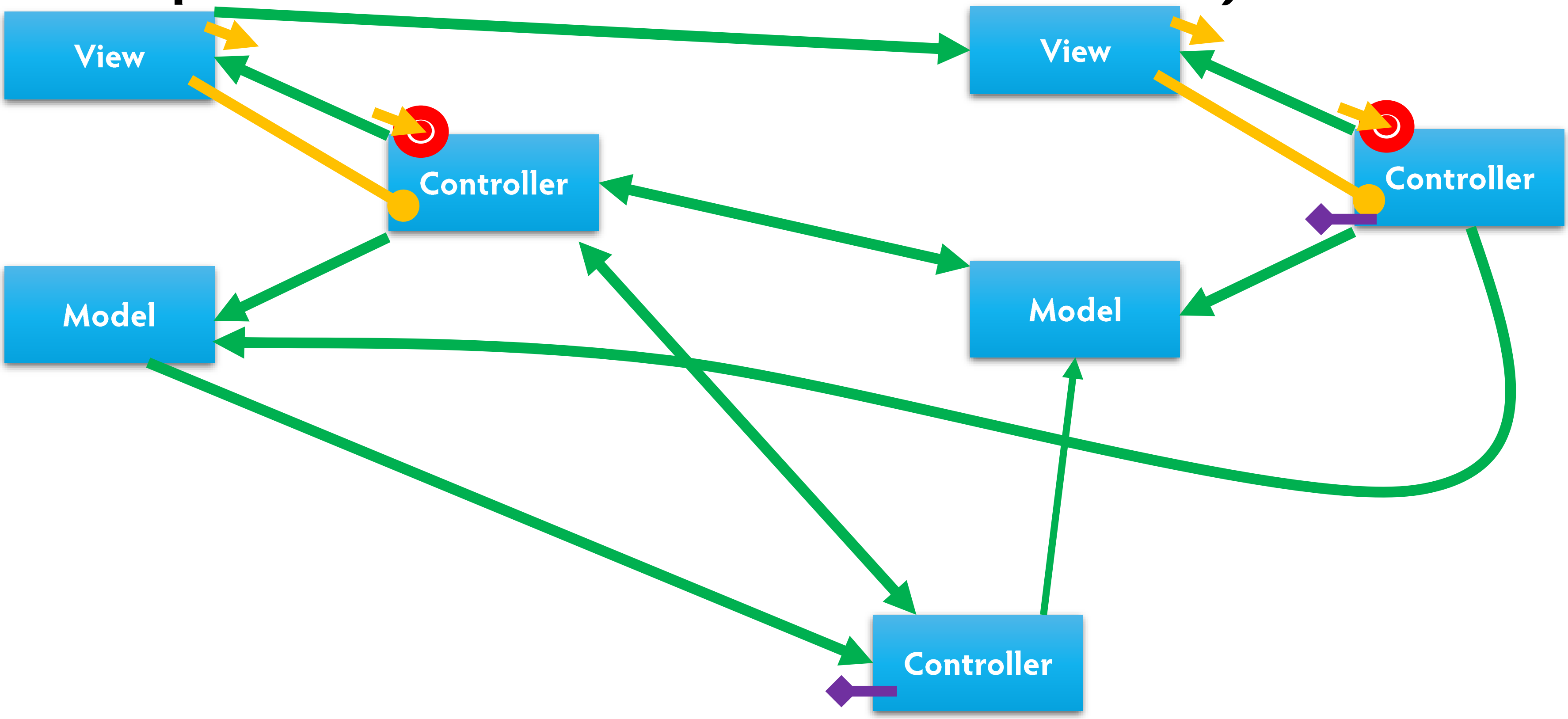
- Model also have to notify the Controller
  - No direct communication allowed
  - Broadcast messages are sent
    - Controllers, can act on



Multiple MVCs – rules are obeyed



Multiple MVCs – rules are disobeyed



Homework – Deadline 11/05 10.15 am

- Create a brief demonstration application
  - Have a Storage class, with internal variable to store values in an array
  - Create functions to retrieve
    - Value at index
    - Most frequent item
    - Smallest/largest item
  - Create a main to test your class, with messages

GNUStep setup

- On Windows 10
  - Install linux subsystem https://docs.microsoft.com/en-us/windows/wsl/install-win10
  - Continue with next block

On Linux

- Install gcc and extensions:
  - sudo apt install gcc gobjc++ gnustep gnustep-devel gnustep-make
- Write your code and compile

```
gcc -MMD -MP -DGNUSTEP -DGNUSTEP_BASE_LIBRARY=1 -DGNU_GUI_LIBRARY=1 -DGNU_RUNTIME=1 -DGNUSTEP_BASE_LIBRARY=1 -fno-strict-aliasing -fexceptions -fobjc-exceptions -D_NATIVE_OBJC_EXCEPTIONS -pthread -fPIC -Wall -DGSWARN -DGSDIAGNOSE -Wno-import -g -O2 -fgnu-runtime -fconstant-string-class=NSConstantString -I. -I /usr/include/GNUstep -o a.out main.m  -lobjc -lgnustep-base
```

Xcode and Android Studio

After the break