
Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

# Basics of Mobile Application Development

OOP and basics of programming techniques


Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

# OOP
# Concepts and practice

# Modelling the world

- Principles
  - Abstraction
    - The properties and parts of the real world are simplified, thus only the essential parts are considered in order to reach the objective which has been set.
    - One abstracts from the unimportant properties and information and the important details are highlighted
  - Differentiation
    - Objects are the entities of the world which have to be modeled.
    - Objects are differentiated based on their important properties and behavior.

# Modelling the world

- Principles
  - Classification
    - Object are assorted into categories, classes. Objects with similar properties belong to the same class, and objects with different properties are in different classes.
    - The classes bear the characteristics of the objects in the class. They can be considered as the templates of the objects.
  - Generalization, specialization
    - Similarities and differences are sought in order to create general or special categories and classes.

# OOP – principles

- OOP principles (Benjamin C. Pierce)
  - Dynamic binding
    - In case of an object, if there are several implementations of a method, the executed one is selected runtime, dynamically.
  - Encapsulation
    - Data and operations are considered as a single unit
    - Practically it is consistent with the definition of type
  - Subtype polymorphism
    - A typed variable can refer to objects with different (other) subtypes
    - Subtype
      - A type created by specializing an existing one
  - Inheritance, or delegation
    - It is possible to create a new class using an existing one
    - It has the properties of the original class
    - And it also can extend, augment and modify the original class
  - Open recursion
    - Special variable, which enables a method to access the current instance of the class

# 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

# OOP – Object

- Object
  - Its state is defined, information is stored
  - Perform tasks, its state can be changed
  - Communicate with another object by messaging
  - Can be identified unambiguously
- Lifecycle
  - Is born – construction and initialization
    - Initial values
    - Tasks executed for initialize
    - Setting the type invariant
  - Exists – operational phase
  - Dies – destruction
    - Freeing resources

# OOP – fields and methods

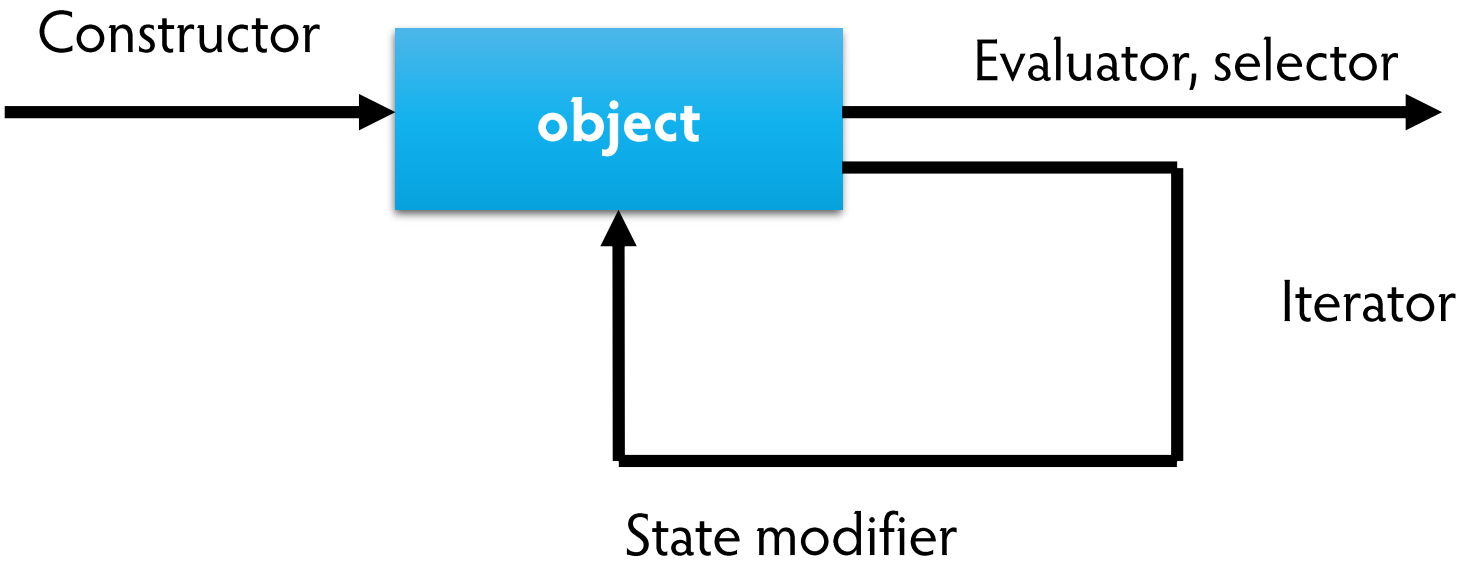
- Class definition
  - Instance variable
    - Separate instance exists for different objects
  - Instance method
    - Works on the state of an instance
  - Class variable
    - Variable for a class
  - Class method
    - Works on the state of the class

# Operations of objects

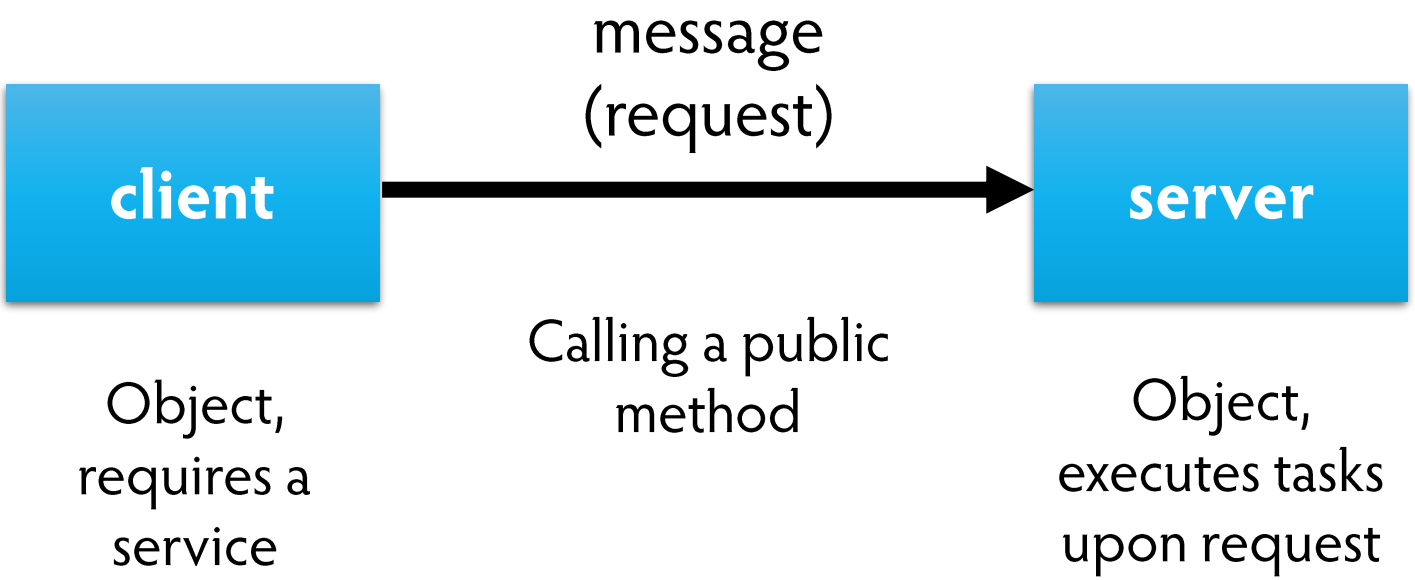
- Export operations
  - Called by other objects
- Import operations
  - Called by the object to provide its defined service
- Operations can be sorted as follows
  - Constructor: to create an object
  - State modifier
  - Selector: to select (except) a part of an object
  - Evaluator: to query features of an object
  - Iterator: to discover (or roam)

# Operations of objects

- Export operations



# Client sends a message



# Client sends message to server

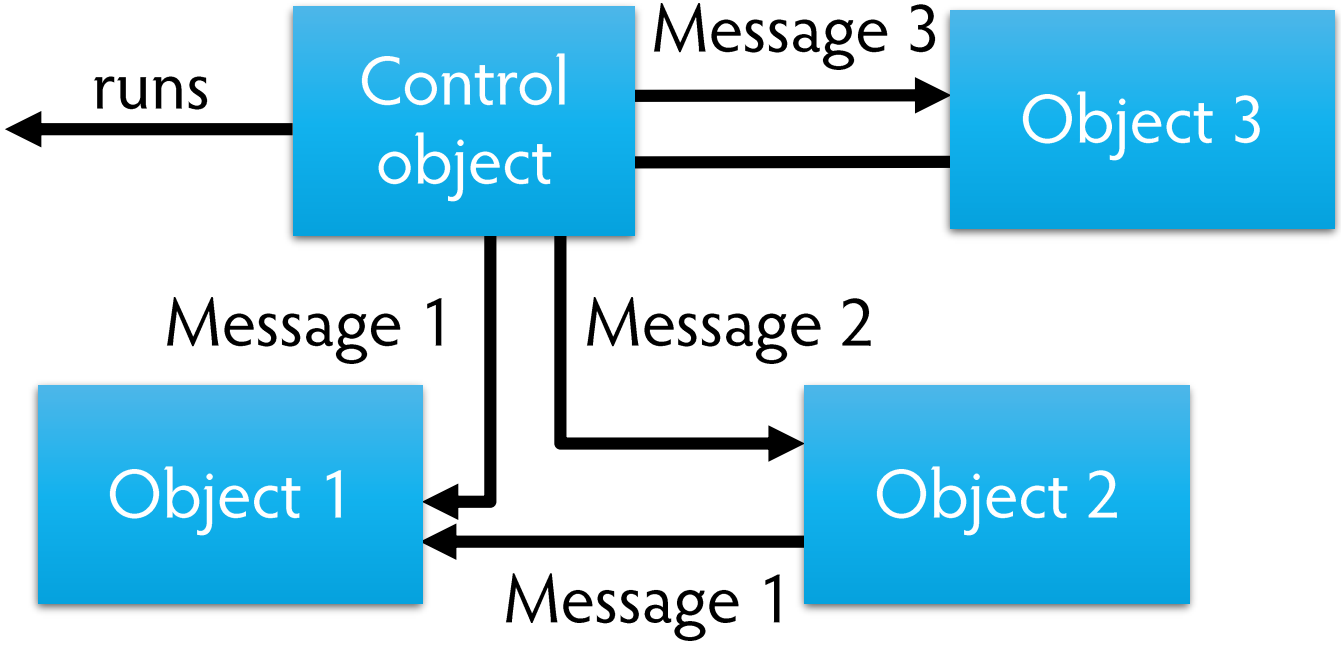
- Client
  - Active object, perform operations on other objects, but operations are not performed on it
  - Do not have any export interface
    - E.g.: Clock – performs an operation regularly
- Server
  - Passive object has only export interface
  - Waits for messages, do not require services
  - Do no have any import interface
- Agent
  - General object, both with import and export interface

# `this`

- Separate memory allocation is made for instance variables
  - Instance methods are working in this state-space
  - However, instance methods are stored only once
  - So, how do we know the which one is the actual instance which is calling a method?
  - We need a pointer which refers to the actual instance, in any of the methods
    - The parameter „**`this`**" is used for this purpose
    - It means that in case of an object wants to send a message to itself; then it should call **`this`**`.message(parameters)` form.
    - So, in all methods, the reference for the actual instance should use this variable.
      - However, it is the default in several languages

# OO program

- An OOP program is a set of communicating objects.
- Each object has its purpose, authority and scope of duties.



# OOP – operational expectations

- Encapsulation
  - Data and operation performed on them considered a single unit
- Information hiding
  - The „private matters" of an object can only be accessed through methods
    - In case of some language this mechanism can be bypassed (not recommended)
- Code reuse
  - The code can be used to create
    - New instance with additional functions
    - New class with new, additional function, or to change existing behavior

# OOP – Inheritance

- Creating a new class based on an existing one
  - The existing methods and fields are used to create new functionality
  - Augment (extend) a class
  - Override existing methods and field, according to the new function
- Design level step
  - Creating a subtype
    - IS-A type relation
    - One can create a HAS-A relation, however, it is not subtype
- Code reuse

# OOP (and not OOP) – Definition

- Overload
  - Two methods with the same name, but different signature
    - The number or type of the parameters are different
    - `int add(int a, int b)`
    - `double add (double a, double b)`
- Override
  - Derived class has a method with the same name and signature
    - If and only if dynamic binding occurs as well, otherwise it is hiding
    - Some of the languages, you have to use a keyword.
      - Objective Pascal: **`virtual`**, **`override`**
- Hiding
  - Derived class has a method with the same name and signature
    - But there is no dynamic binding
  - Fields and variables can hide each other in a block as well as in a child class

# Abstract class

- Design tool
  - To generalize
  - Subtype relation often requires creating abstract classes
- Abstract class: incomplete class
  - There are functions which implementations are not known in the class
  - The implementation is provided by one of the derived classes

# Polymorphism, dynamic binding

- Polymorphism is the capability when a variable can refer to different type of objects
  - Now we consider only the subtype polymorphism
  - Type A is the subtype of type B if the following is true: type A can be used in all situations where type B can be used
  - Static type: defined at declaration
  - Dynamic type: the type the actual object referred by the variable
  - The dynamic type can be the static type of any of its derived classes.
  - The static type is permanent, and set in the source code, while dynamic type can vary in runtime
- Dynamic binding
  - The dynamic call is the event when we call a method of an object allowed by its static type, but the implementation executed which corresponds to the dynamic type.
  - Dynamic binding happens only when the derived class overrides the method (not hiding)

# C++

# Class definition

```
class aclass : public parent_1, public parent_2 {
   // Fields
   private:
      int counter = 0;

   public:
      aclass();
      void add(int howmany);
      void print() const;
}
```

```
aclass::aclass(int start) { counter = start; }
```

```
void aclass::add(int howmany)
      {counter += howmany; }
```

```
void aclass ::print const { cout << counter; }
```

# Keywords

- Fields
  - **`public`** – all objects have access (this, children, others)
  - **`protected`** – this and children objects can access
  - **`private`** – only this class can access (and friends)
  - **`const`** – constant variable, its value cannot be changed
  - **`static`** – class variable (can be accessed without instantiation)
- Methods
  - **`public protected private static`**
  - **`const`** – does not change the state of the class
  - **`void`** – when there is no return value
  - **`virtual`** – functions with dynamic binding (overriding instead of hiding)
  - After signature `=0;` – pure virtual, abstract function
  - **`throw`** – throwable exception can be listed

# Type and passing parameters

- In C++ regular variables and pointer as passed by value
  - Formal parameters are declared as local variables
  - They are initialized with the values of the actual parameters
    - Regular variable holds the value which is assigned
    - In case of a pointer, this is the memory address
  - Thus a copy is created of the original variable to an other part of the memory
  - As a result, the formal and actual parameters are in different places
- In C++ a reference formal parameter declares a (new) reference to the actual parameter
  - The original memory location has a new variable name
  - All of the changes performed through the formal parameter (local variable) effects the original memory location

# Constant parameters

- It can be avoided that the called function can change the value of the original value
  - Then the parameter is constant
  - `const int & i`
  - You can pass large objects without copying them as well as you can prohibit any of changes
- It works with pointers as well
  - `void f(int * const p)`
    - The address is constant (as that is the value of the variable)
  - `void f(const int * p)`
    - You cannot change the referred memory
    - `void f(int const * p)` is equivalent
  - `void f(int const * const p)`
    - Both address and referred memory is constant

# Inheritance

- The graph representation (directed graph) of the inheritance relations can be called as class hierarchy
- In C++ multiple inheritance exists.
  - Thus the graph is a general directed graph
- Inheritance can be
  - **`public`** – Visibility modifiers are unchanged, but **`private`** members cannot be accessed in derived classes
    - IS-A relation
  - **`protected`** – Public functions and fields will be **`protected`**
  - **`private`** – Public functions and fields will be **`private`**
- Default is **`private`**

# Multiple inheritance

- A class can inherit (directly) from several other classes
- What happens if there are functions with the same name in different parents?
  - A decision must be made
  - Scope operator
    - `d.Base1::f();`
    - `d.Base2::f();`
- Diamond problem
  - Virtual inheritance
    - `class C: public virtual Base`

# Constructor

- Constructor
  - Code, which is executed automatically when the class is instantiated
  - Its name is the same as the class, and it has no return value
  - It is similar to methods, but there are differences (it is not member, because it cannot be inherited)
  - All classes has constructors
    - If we do not define, the compiler creates
    - But only when there is no programmer defined constructor
  - Several constructors can exist, the can be overridden
  - The constructor of a derived class called after the constructor of the base class
  - In C++11 constructors can be delegated
    - Avoid cyclic call

# Destructor

- Destructor
  - Destructor is a code which is responsible to free resources most of the cases
    - Prepare to die
  - Classes should have virtual destructors
    - If there is no chance to have a derived class it is not necessary

# Friend

- A function can access to any fields of a class
  - Even private
  - Encapsulation can be violated
  - Keyword **`friend`** have to be used
  - A friend can access the private and protected parts of a class
  - Typical examples are the input/output stream operators (<<, >>)

```
friend std::ostream& operator <<
(std::ostream& stream, const Object& z);
```

# `delete` and `default` functions

- Complier creates a bunch of functions, if they are not defined manually
- Starting from C++11 this automatism can be controlled.
  - To create use **`default`**
  - To avoid use **`delete`**
- ```
  class Car {
        public:
              Car() = default;
              Car(const _car&) = default;
              Car& operator=(const _car&) = delete;
              virtual ~C() = default;
  };
  ```
  ```

# `delete` and `default` functions

- The class-level operators (`&, *, ->, new, delete`) also can be controlled.
- 
```
Class C {
    public:
        void *operator new(size_t) = delete;
        void *operator new[](size_t) =
delete;
};
```
- 
```
int main() {
    C *c = new C;
    C *t = new C[3];
    C _c;
    C _t[10];
}
```

# Assignment operator, copy constructor

- They can be used to copy an object
- Assignment operator is called when the variable is assigned to a new (existing) value
- Copy constructor called when the parameter is passed by value
- The are declared automatically
  - In case of dynamic memory allocation, the automatically generated assignment operator and copy constructor cause shallow copy
    - As only the pointer is copied not the referred memory
  - As a results they must be declared manually

# Assignment operator, copy constructor

```
class A {
        A (const A& _other);
        A& operator= (const A& _other);
        A (A&& _other);                 // move constructor
        A& operator= (A&& _other); // move operator
};
A e, f, g;
g(std::move(f));
e = std::move(g);
```

- http://en.cppreference.com/w/cpp/language/move_constructor
- http://en.cppreference.com/w/cpp/language/move_operator

# User defined literals

- As of C++11 user can defined new literals

```
inline double operator"" _deg (long double degree) {
  return degree * 3.14159265 / 180.0;
}
double rad = 90.0_deg; // degree = 1.570796325


unsigned operator"" _Magic(const char* _magic) {
  unsigned b = 0;
  for(unsigned int i = 0; _magic[i]; ++i)
     b = b*2 + (_magic[i] == '1');
  return b;
}
int mask = 110011_Magic; // mask = 51
```

# Java

# Class definition

```
public class AClass extends Parent implements Interface
{
  private int counter;

  public AClass()
  {
       counter = 0;
  }

  public void add(int howmany) throws MyException
  {
       this.counter += howmany;
  }


  public void print()
  {
       System.out.println(counter);
  }
}
```

# Keywords

- Class
  - **`public`** – everyone can access
  - **`final`** – cannot be derived
  - **`abstract`** – abstract class, with abstract method(s)
  - **`extends`** – to provide the parent class
  - **`implements`** – to list the implemented interfaces
- All of them are optional. The default visibility is package private – it means that the class is accessible in the package only.
- Fields
  - **`public`** – everyone can access (this class, derived class, inside and outside of the package)
  - **`protected`** – can be accessed in this class, derived classes and in the package
  - default visibility – package private, can be access inside of the package (this class, and …)
  - **`private`** – only this class
  - **`final`** – its value cannot be changed, constant
  - **`volatile`** – always committed to the shared memory in case of threading
  - **`static`** – class variable (can be accessed without instantiation)

# Keywords

- **`final`** – again
  - What is constant? The value:
    - In case of primitives it is the value
    - In case of `Object` it is the reference to the object
  - It has no effect on the fields of the referred object
- Methods
  - **`public protected private abstract static`**
  - **`final`** – cannot be overridden or hidden
  - **`synchronized`** – mutual exclusion can be achieved in threading
  - **`void`** – if there is no return value
  - **`throws`** – the throwable exception must be enumerated

# Visibility

| Originating object | | Target object | | | |
|---|---|---|---|---|---|
| Package | Class | public | protected | no modifier default | private |
| Same | This | X | X | X | X |
| | Inner | X | X | X | X |
| | Derived | X | X | X | |
| | Other | X | X | X | |
| Other | Derived | X | X | | |
| | Other | X | | | |

# Types, parameter passing

- In Java parameters always passed by value
  - Thus the value is copied
  - In case of reference type the value is the memory address, so the address is copied this the parameter passing seems „by reference"
    - However the wrapper counterparts are immutable, as a result the behave as the primitive ones
- Primitives:
  - `byte, short, int, long, float, double, char, boolean`
- Everything other is reference type, derived from Object class
- Primitive – Wrapper pairs
  - `Byte, Short, Integer, Long, Float, Double, Character, Boolean`
    - Immutable – new instance is created once it is changed
- Warning! `String` is immutable

# Inheritance

- The graph representation (directed graph) of the inheritance relations can be called as class hierarchy
- In Java there is an universal base class, the `Object`, everything is derived from `Object`
- In Java there is no multiple inheritance, thus the class hierarchy is represented by a tree
- Implicit `extends Object` in case of no manual `extends` given
- `Object`: pre defined in `java.lang`
  - There are methods required to exist in all objects

# Initialization of an object

- The order of execution
  - Static initialization block of the base class
  - Static initialization block of the derived class
  - Initialization block(s) of the base class
  - Constructor of the base class
  - Initialization block(s) of the derived class
  - Constructor of the derived class
- The constructor of the base class is executed before any constructor of the derived class
  - Even if it is not called explicitly
    - In that case the constructor with same signature is called
    - If it does not exist then an Exception is thrown
- The constructor without parameters is created by the compiler if and only if there is no other constructor is defined

# Initialization block

- Initialization block: Block of statements (instance level and class level as well) to initialize variables

```
public class A{
        static int i=10;
        static int ifact;
        static {
                ifact=1;
                for (int j=2; j<=10; j++){
                        ifact*=j;
                }
        int [] array = new int[10];
        {
                for (int i=0; i<10; i++){
                        array[i]=(int)Math.random();
                }
        }
  }
```

# Initialization block

- Class level: executed when the class is initialized, substitutes the class constructor (as they do not exist)
- Object level: executed when the object is instantiated, augmenting the constructors.
  - In case of anonymous classes, it substitutes the constructors
- There may be several initialization blocks in a class
- Execution order is the order of the definition (merged with the variable initializations)
  - Variables defined later cannot be referenced
- `return` statement is not allowed

# Java – implicit type conversion

- Subtypes
  - Subtypes can be handled as general types
  - So a derived class can be handled as its ancessors
- The object is not converted
  - `java.lang.ClassCastException` is raised when the type is not compatible
  - Use **`instanceof`** operator!

# Interfaces

- Interfaces can be implemented by classes
- Keyword
  - `public class Apple implements Edible`
  - If a class implements an interface then the interface can be used as static type
- Static type vs dynamic type:
  - ```
    Edible apple = new Apple();
    Edible pear  = new Pear();
    ```
  - Supposing that
    - ```
      Apple implements Edible
      Pear  implements Edible
      ```
- Interfaces can be inherited
  - All the constants and member functions can be inherited from the superclass
    - Implementation not, but that cannot be there.
- Multiple inheritance is possible!
  - As interfaces are „abstract" there is no problem with the inheritance of implementation

# Modifiers

- Functions
  - Methods are always **`public abstract.`**
    - These keywords are optional
    - Using others is error
- Constants
  - All fields are **`public static final`**
  - Not required
- What are the differences between abstract class and interfaces?

# Homework – Deadline: 10/01/18 10.15 am

- Create a C++ program which demonstrates the proper and improper usage of friend
  - Read first!
  - Defining friend operators might be appropriate!
  - Violating the encapsulation and data hiding is inappropriate!
- Create a Java demonstration where it is a good option to use default interface methods.
  - Read first!
  - Also, demonstrate why it should be considered as the last resort!

# Programming patterns

Next week