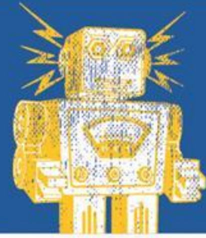


Selected Topics in Java

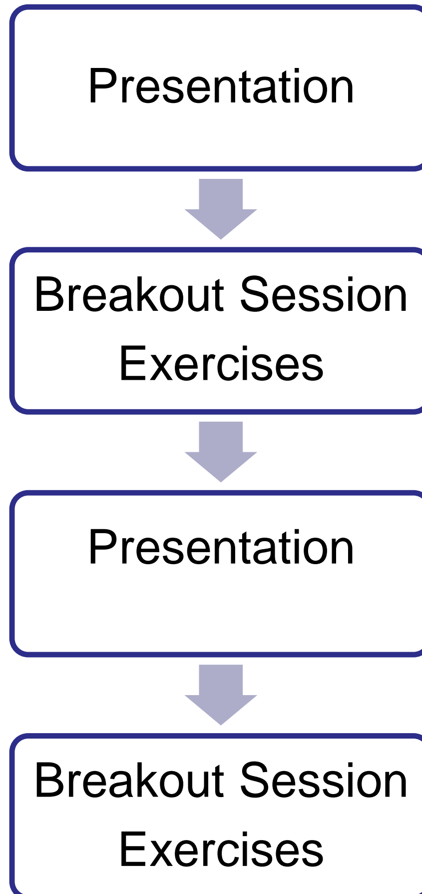
OOP Advanced, Factory Pattern

Martin Deinhofer, WS2018

© FH Technikum Wien

Course Overview

- Structure of a block



Breakout-Sessions

1. **Duration: ~30-45 min.**
2. **Exercises can be done in groups of 2**
3. **Solution presented by 1 group at the end of a breakout session**

4. **Reference solutions can be found on https://es.technikum-wien.at/embedded_systems_public**

© FH Technikum Wien

FH University of Applied Sciences TECHNIKUM WIEN

Object Oriented Programming
Interfaces, Abstract classes

© FH Technikum Wien

FH University of Applied Sciences TECHNIKUM WIEN

Interfaces

- An interface
  - is a **specification** of a contract for a class
  - **declares methods** but does not implement them
  - does not have constructors and member variables
  - allows for code abstraction

- Classes can implement interfaces
- Classes must implement all methods declared in the interface

© 2012/13 UAS Technikum Wien

Interfaces – Declaration & Implementation

```
interface Moveable
{
        public void move();
}

class Piano implements Moveable
{
        public void move() {
                System.out.println("Heavy lifting");
        }
        public void play() {
        }
}
```

Defines method signature, hides implementation details

Many classes could implement interface Moveable

© 2012/13 UAS Technikum Wien

Interfaces – Usage

```
…
public static void main(String[] args) {
        Moveable moveAbleImpl=new Piano();
        moveAbleImpl.move();
        moveAbleImpl.play(); //won‘t compile

        Piano pianoImpl=new Piano();
        pianoImpl.move();
        pianoImpl.play();
}
…
```

© 2012/13 UAS Technikum Wien

Abstract classes

- An abstract class is a mix of a class and an interface
- An abstract class
  - **cannot be instantiated**
  - does not need to implement all methods it declares
  - allows the implementation of common methods while other methods can be implemented in subclasses

© 2012/13 UAS Technikum Wien

Abstract classes

```
public abstract class Instrument
{
        String name;
        public String getName()
        {
                return name;
        }

        public abstract void play();
}
```

Subclass inherits implementation, but may override it

Implementation must be done by subclass

© 2012/13 UAS Technikum Wien

Abstract classes

```
public class Piano extends Instrument {

     @Override
     public void play() {
         System.out.println("I am playing piano...");
     }

}
```

```
//does not compile
//Instrument instr=new Instrument();

Instrument instr=new Piano();
//getName is already implemented by abstract class Instrument
System.out.println(instr.getName());
//Method play is implemented by concrete subclass. Due to abstract declaration the
caller can be sure that the method exists!!
instr.play();
```

© 2012/13 UAS Technikum Wien

Final classes

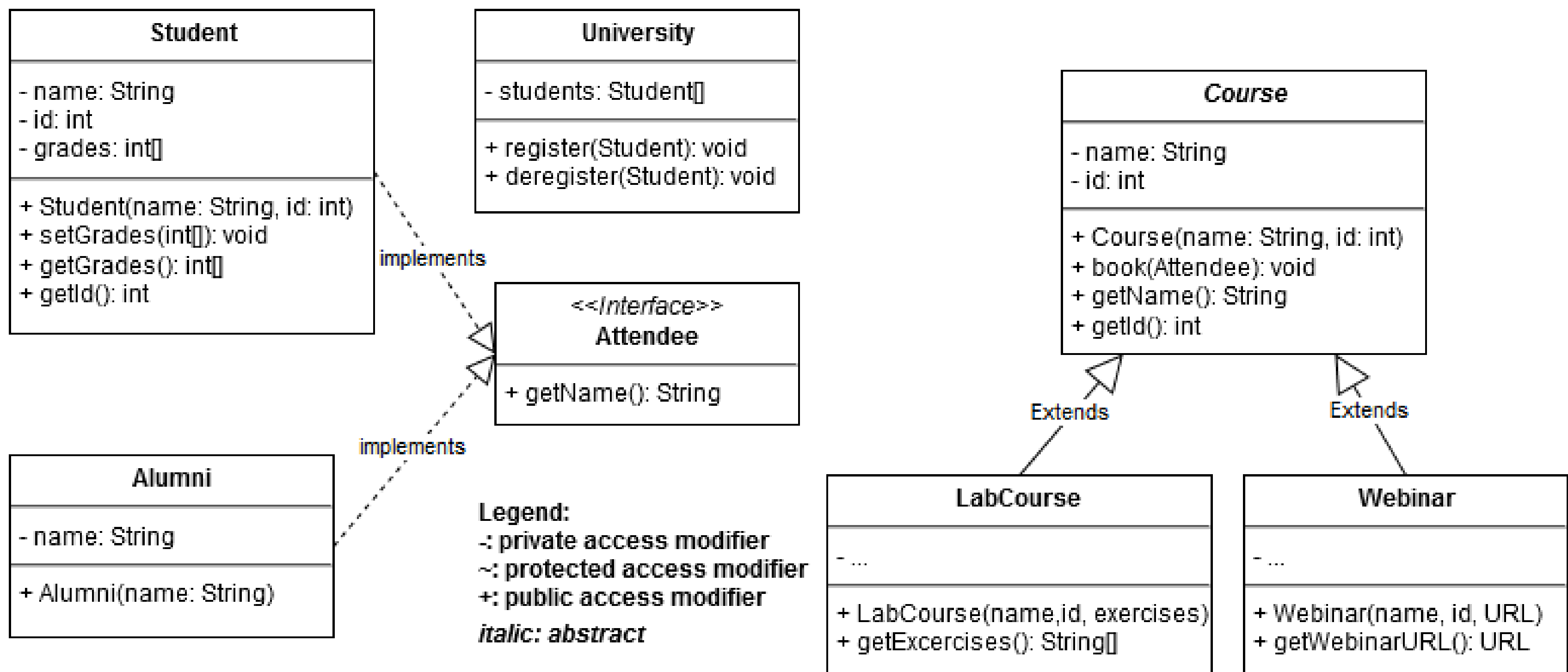
```
final class Instrument
{
        String name;
        public String getName()
        {
                return name;
        }
}
```

- A final class cannot be extended by inheritance
- → The behaviour of the class can not be changed by the user of the class.

© 2012/13 UAS Technikum Wien

1. Breakout – Exercise Suggestions

1. Model following classes and instantiate them



© 2012/13 UAS Technikum Wien

1. Breakout – Exercise Suggestions (cont.)

- Class *Course*:
  - Which methods are
    - inherited?
    - overridden?
  - Are the member variables *name* and *id* accessible by objects of *Student*, *Alumni* or *University*?
- Class *Alumni*:
  - Can alumnis be registered at the university?
  - Are alumnis allowed to book a course?
- What do alumnis and students have in common?

© 2012/13 UAS Technikum Wien

Design Patterns
Factory Pattern

© FH Technikum Wien

Design Patterns - Definition

- A design pattern is a …
  - “…general applicable and reusable solution that solves frequently occurring problems and architectural challenges in software design.”
    (Baesens, Bart (2015); Beginning Java Programming)

© FH Technikum Wien

Design Patterns - Definition

- Borrowed from Architecture (1977)
- Famous book of „**Gang of Four (GoF)**“ (Gamma, Helm, Johnson, Vlissides (1994)):

  „Design Patterns: Elements of Reusable Object-Oriented Software”

  Defined 23 SW Design Patterns

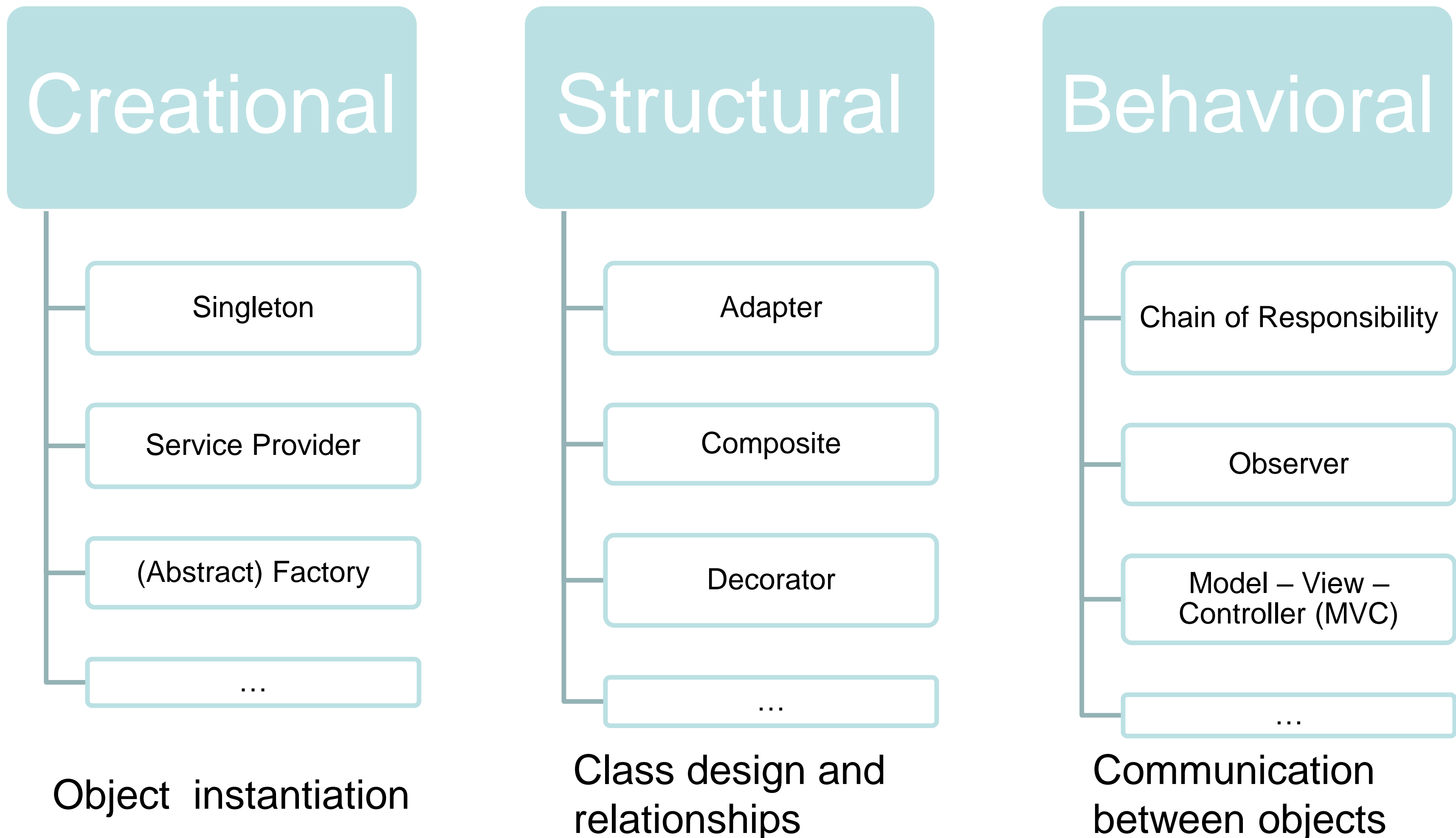
Benefit of Design Patterns

- Common terminology for developers
  - When talking about design ideas developers can refer to the design pattern terms
    e.g. If a class should only have one instance at maximum
    → "We need a Singleton pattern!"

- Best practices solutions
  - Reusing best practices generally improves
    - maintainability
    - code flexibility for changes
    - code understanding

© FH Technikum Wien

Design Patterns - Categorization



© FH Technikum Wien

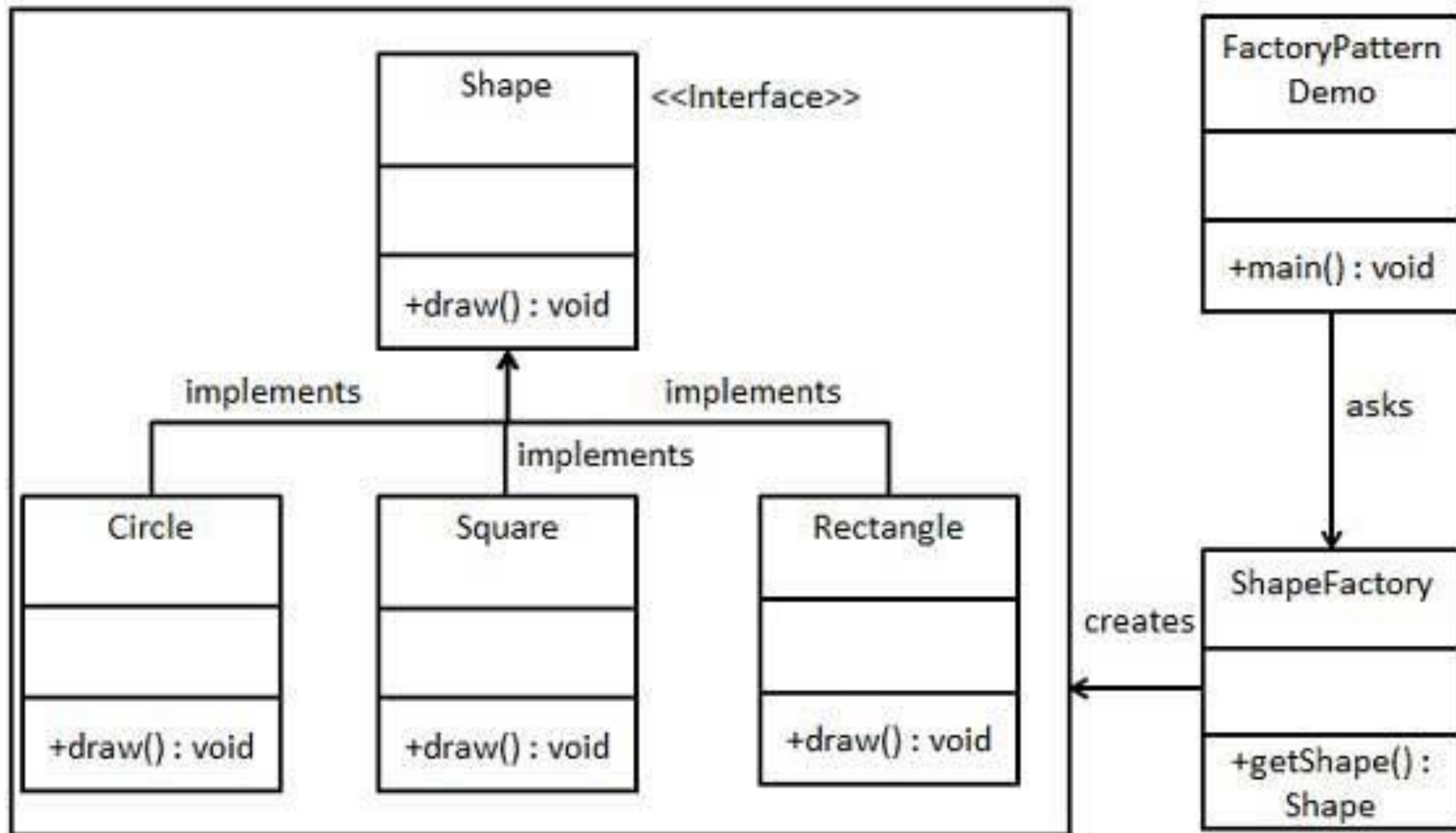
Factory Pattern

- Definition: Defer instantiation of a class to subclasses
  → Defines interface for object creation
  → Factory classes provide different implementations for instantiation

- Pros
  - **Seperation** of **object creation** and **object usage**
    → creation can be changed without impact for other code
  - Central point for object creation

© FH Technikum Wien

Factory Pattern example



Factory Usage Example

```
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();
        //get an object of Circle and call its draw method.
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        //call draw method of Circle
        shape1.draw();

        //get an object of Rectangle and call its draw method.
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        //call draw method of Rectangle
        shape2.draw();
    }
}
```

© FH Technikum Wien

Behavioral Patterns

- Observer
  - Observer pattern is used when there is one-to-many relationship between objects,
  - such as if one object is modified, its dependent objects are to be notified automatically.
- Model-View-Controller
  - **Model** represents an object carrying data. It can also have logic to update controller if its data changes.
  - **View** represents the visualization of the data that model contains.
  - **Controller** acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

© FH Technikum Wien

2. Breakout – Exercise Suggestions

1. Execute the step by step instructions of the Factory Pattern example: Java Design Patterns - Problem Solving Approaches, Chapter 2, Page 3

   - Create interfaces and classes
   - Create demo code to use factory and invoke *draw* method
2. What if you need to parametrize (radius, length,…) shapes?

© FH Technikum Wien