



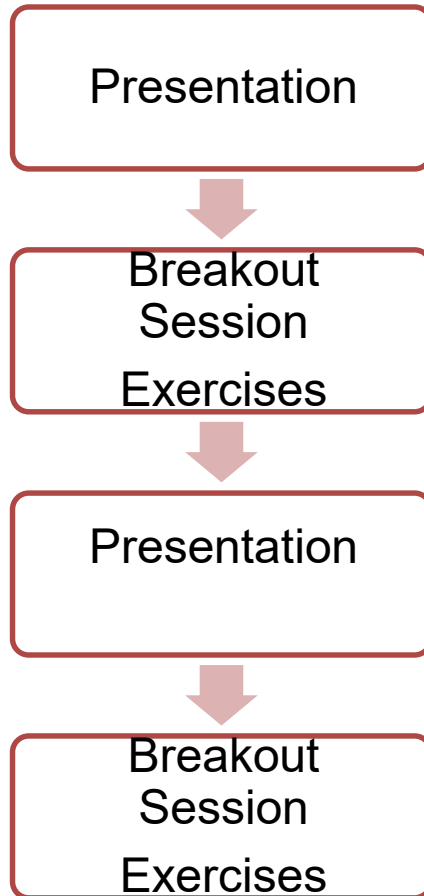
Selected Topics in Java

OOP1 (classes, inheritance), UML

Martin Deinhofer, WS2018

# 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**

1. **Reference solutions can be found on [https://es.technikum-wien.at/embedded\\_systems\\_public](https://es.technikum-wien.at/embedded_systems_public)**

# Object Oriented Programming (OOP) Classes

# Object Oriented Programming

- Basic idea? - Represent the real world objects and their interactions
  - Baby
    - String name
    - boolean isMale
    - double weight
    - double decibels

# Object Oriented Programming

- Why objects? Why not just primitives?

```
// baby alex
String nameAlex;
double weightAlex;
// baby david
String nameDavid;
double weightDavid;
// baby david2
String nameDavid2;
double weightDavid2;
```

Someone would have to create a variables for each attribute and instance:

**nameAlex1...nameAlex{n}**  
**weightAlex1...weightDavid{n}**

or create **arrays** for each attribute

- Scalability!

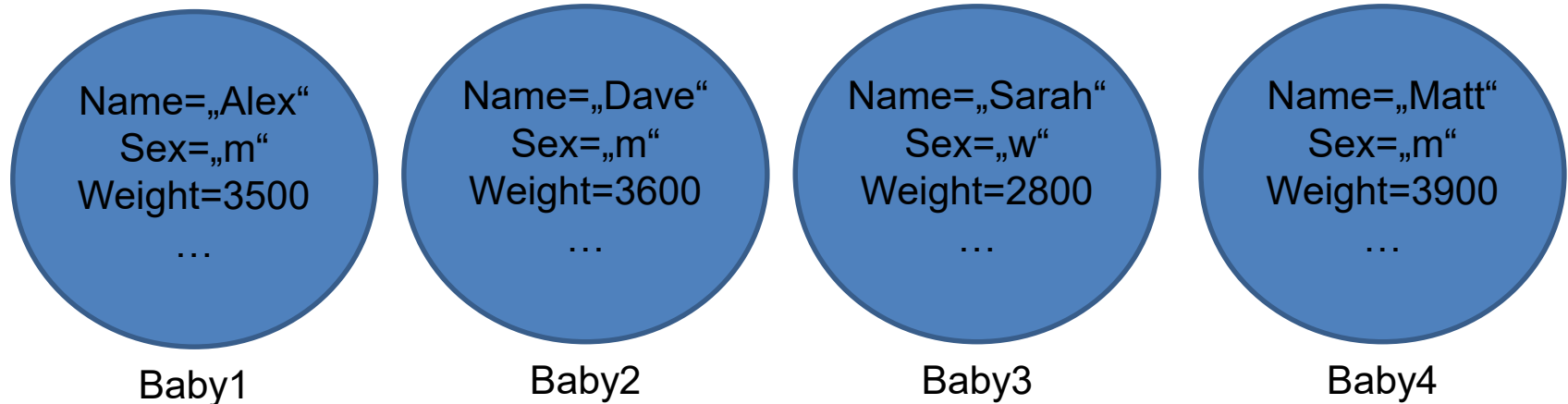
# Object Oriented Programming

- Why objects?



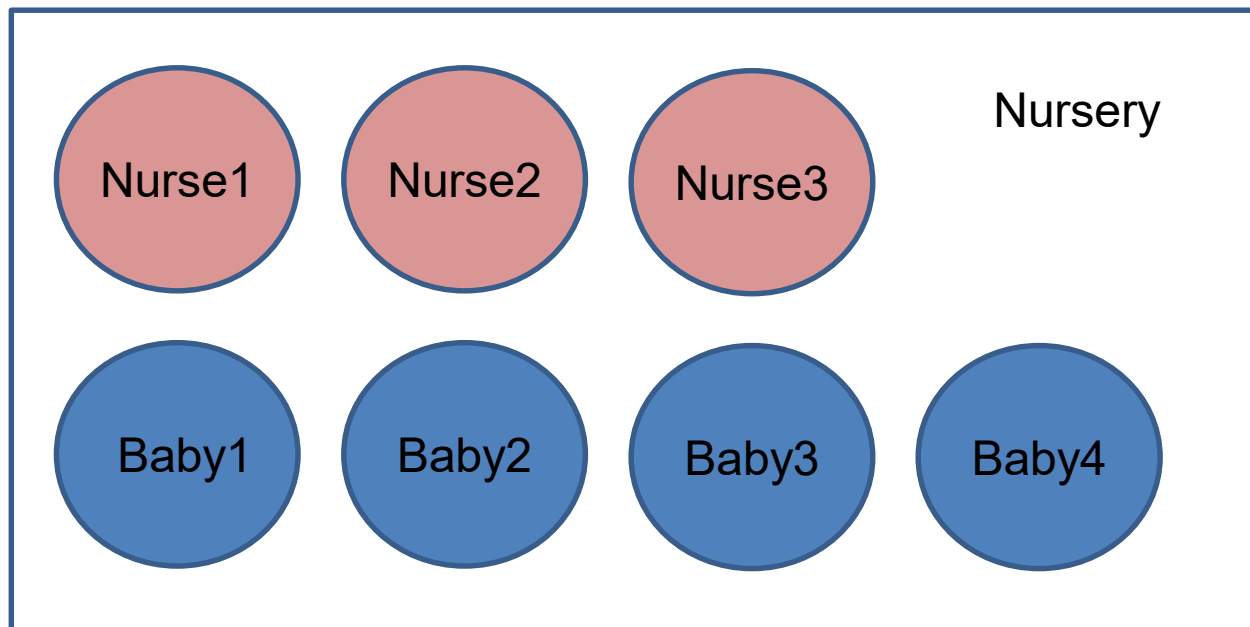
# Object Oriented Programming

- Why objects?



# Object Oriented Programming

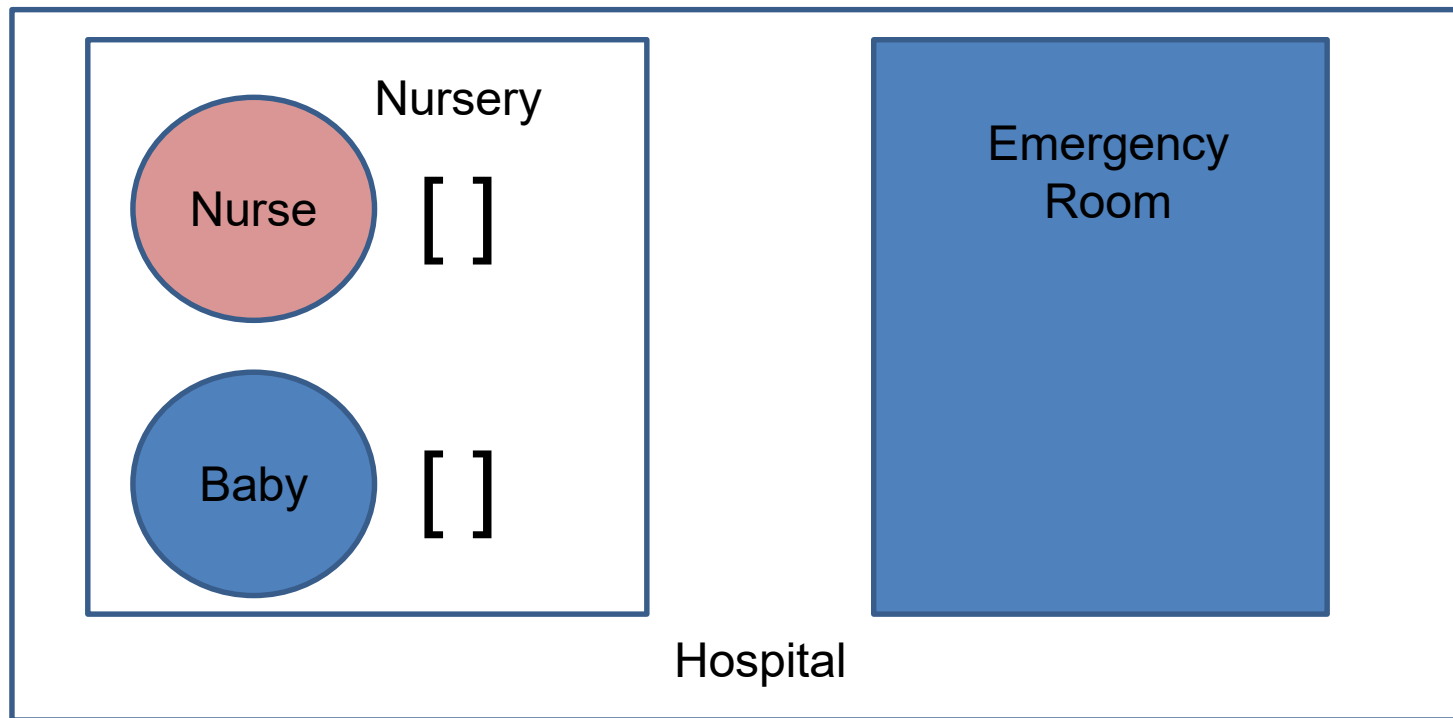
- Why objects?





# Object Oriented Programming

- Why objects?

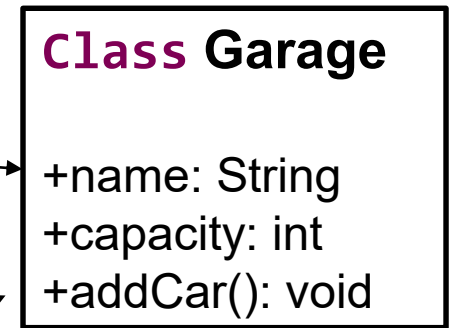


# Object Oriented Programming

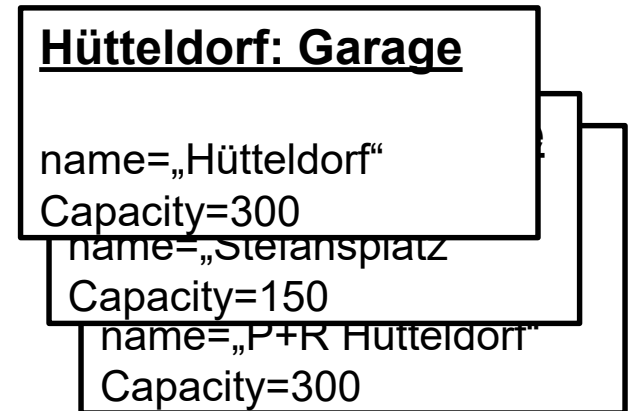
- Why objects?
  - model the real world including
    - data structure
    - attributes
    - relations (associations)
  - allow modelling of systems of vast complexity
  - break up complex system into smaller simpler parts
  - facilitate extending of a system
  - Encapsulate data or code (to protect it from unintended use) -> blackbox

# OOP - Definitions

- **Class** – a template of a data object
  - **Member Variable, Field** – variable representing an attribute or property of the class
  - **Method** – a sequence of instructions
  
- **Object** – an instantiation of a class, physically represented in memory



Many instances



# Object Oriented Programming

- **Classes/“Objects“** represent data structure and code:
  - **Data structure** of
    - Primitives (int, double, char, etc..)
    - Objects (String, Integer, Double, Array, MyClass, etc...)
  - **Code**
    - Methods, Constructors,...
- **C** knows
  - **functions** for code
  - **structs** for data

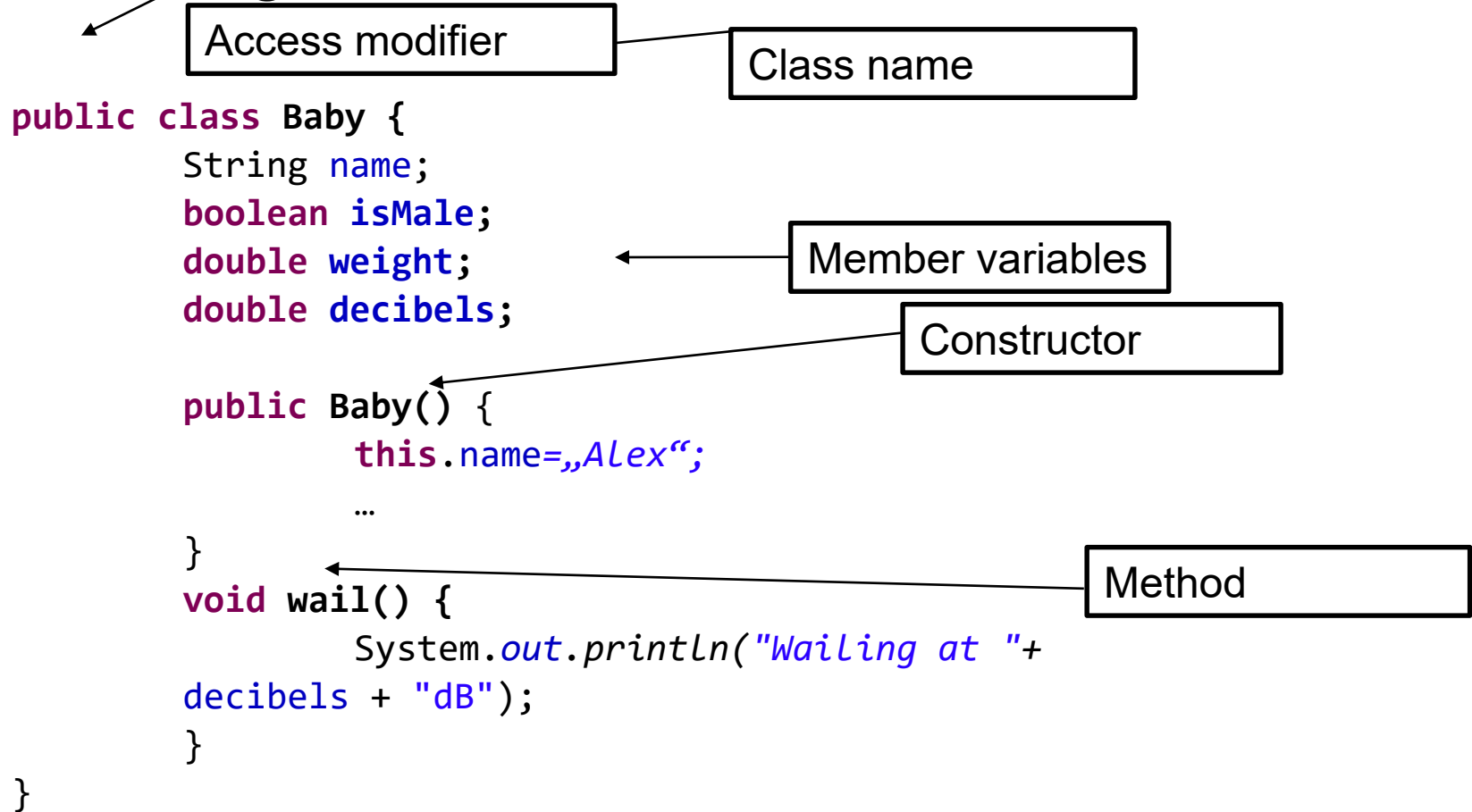
# Summary of Characteristics of a Class

- is a template for objects
- defines common characteristics
- defines meta information
- is the “data-type” of an object
- has fields (member variables, attributes)
- has methods (operations)
- has modifiers (private, default, protected, public)
- has abstract, final characteristics
- implements an interface
- inherits (extends) a super class
- Overrides method implementations of a super class
- ...

# Defining Classes

```
[access][abstract/final]class className  
    [extends superClassName]  
    [implements interfaceNames...]  
{  
    //fields (member variables)  
    //constructors  
    //methods (member functions)  
}
```

# Defining Classes



# Defining Classes

- Usually a single class is declared in one file
- The **public** class in the file must have the **same** name as the file
- Class names should use CamelCase notation:
  - e.g.: BattleShip, ThisIsMyVeryComplexClass
- If a class has a **main method**, it can be run via the “java” command



# Methods

- Classes can not only hold data
- Classes also provide means to “**send messages**” to an instance of a class - these are called methods
- Similar to C function, but exists within class
- Has access to data within the class

# Methods

- Methods perform functions
- Methods work on the state of the class
- Methods can have multiple arguments, and return up to one value
- If no value is to be returned, use the keyword `void`
- A class can have as many methods as needed

- Template:

```
[access] returnType methodName([arguments...]) {  
    //method body  
}
```

# Creating an Instance of a Class

```
Baby baby1 = new Baby();
```

- New operator tells JVM to create a new instance
- *Baby()* is a call to the constructor of the class
- Variable *baby1* of class “Baby” holds reference to that new instance
  
- What values do the fields hold?

# Creating an Instance of a Class

- Basically fields will hold their default values

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000' (or 0)
boolean	<i>false</i>
Reference types (objects)	<i>null</i>

# Constructors

- Must have the same name of the class that they are in
- Multiple constructors with different parameter list may exist for a class (overloading)
- Method that handles initialization of class
- No return type!
  
- Template:  

```
[access] className([arguments...]) {  
    //constructor body  
}
```

# Constructors

- If no constructor is implemented a **default constructor** without any parameters is provided automatically
- If any class constructor is implemented, there is **no default constructor**
  - If a parameter-less constructor is then needed, it has to be implemented

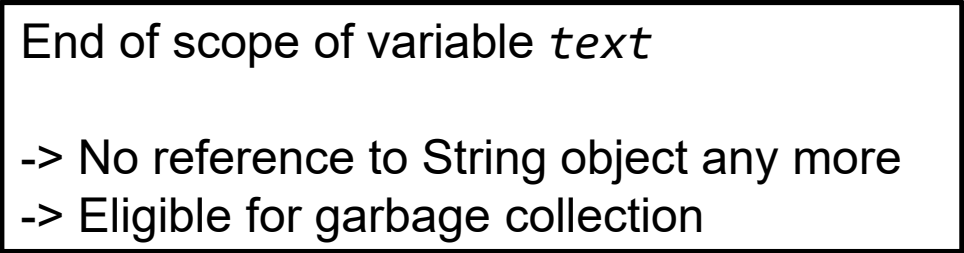

# Where is my destructor?

- Unlike C++, Java does not need destructors
- **If an instance is no longer referenced** (variable's scope ends):
  - > **marked for destruction**
- JVM will release all the instance's data and the instance itself in next run of garbage collector
- However there is a method **finalize()**
  - Can be overridden to do cleanup (close file,...)
- To force garbage collection, set all references to **null**

```
Baby baby1=new Baby();  
Baby baby2=baby1;  
baby1=null;  
baby2=null;
```

# No Destructor → Garbage Collector

```
public class Baby {  
    String name;  
    public Baby() {  
        this.name = „Alex“;  
        wail();  
    }  
    void wail() {  
        String text = "Wailing at " + 20 + "dB";  
        System.out.println(text);  
    }  
}
```





# Non-Static versus Static Elements

- Fields usually represent data that belongs to an instance of a class
- However there can be fields (and methods) that should be shared across all instances of a certain class
- use the keyword **static** for a field or a method declaration
- Access via:  
    [ClassName].[fieldName|methodName]

# Non-Static versus Static Elements

```
public class Bean {  
    public int beanCounter= 0;  
    public Bean() {  
        beanCounter++;  
    }  
    public static void main(String[] args) {  
        new Bean(); new Bean();  
        Bean bean = new Bean();  
        System.out.println(bean.beanCounter);  
        // Prints "1"  
    }  
}
```

# Non-Static versus **Static** Elements

```
public class Bean {  
    public static int beanCounter= 0;  
    public Bean() {  
        beanCounter++;  
    }  
    public static void main(String[] args) {  
        new Bean(); new Bean(); new Bean();  
        System.out.println(Bean.beanCounter);  
        // Prints "3"  
    }  
}
```

# Method Overloading

- Unlike in C (but like in C++) Java classes can have multiple methods with the same name but a **different parameter list (distinct method signature)**
- For example:
  - void test() { ... }
  - void test(int number) { ... }

└──────────────────┘  
method signature
- myClass.test(45); will call the second method

# „this“ Keyword

- Non-static parts of classes can use the keyword *this*
- *this* is a reference to the current instance itself
- Can be used to pass the instance to a method
- Can be used to overcome variable scope collisions

```
public class Bean {
    private int beanCounter= 0;

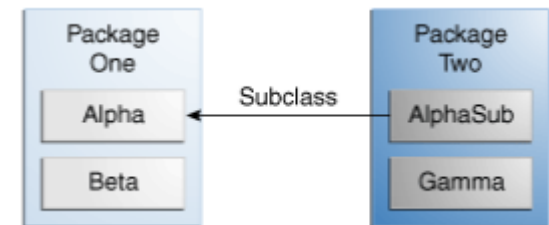
    public Bean(int beanCounter) {
        this.beanCounter =
beanCounter;
    }
}
```

# Access Types

- What is the placeholder [access]?
- There are 4 types of access keywords to describe which classes have access:
  - public – any other class in any package
  - protected – any subclass has access
  - (no modifier) – only classes within the same package, no subclasses
  - private – only accessible from within the same class
- **Information hiding (Blackbox):** Hiding of attribute and implementation details

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N



Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

# Access Types

**public class** Bean {

not visible by caller method of other class

**private int** beanCounter = 0;

public setter method

**public int** setBeanCounter(**int** beanCounter) {  
    **this.beanCounter** = beanCounter;  
}

public getter method

**public int** getBeanCounter() {  
    **return** beanCounter;  
}

}

# Method Call

## Call by Value vs. Call by Reference of parameters

- Call by value - Primitives
  - Called method cannot change value of caller method
- Call by reference - Objects
  - **Attention: Called method can change values of object** (*myCircle*) instantiated in caller method

↑  
 moveCircle(myCircle, 23, 56)

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);
}
```

Values of myCircle after method call?

Answer: (x+23, y+56) }



# Method Call

## Call by Value vs. Call by Reference of parameters

- Call by reference – Objects
- What happens if the called method instantiates a new object?

```
moveCircle(myCircle, 23, 56)
```



Values of myCircle after method call?  
 (x+23, y+56) ?  
 (0,0) ?

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    1 circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    2 // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

Answer: (x+23, y+56)

Order	Reference var.	Object
1	myCircle, circle	x: x+23, y: y+56
2	myCircle	x: x+23, y: y+56
	circle	<b>x: 0, y: 0</b>

## Method call

Special case: Primitive wrapper classes and Strings

- String objects and the wrapper classes for primitives (Integer, Double, ...) are **immutable**
  - any change to them will result in a new instance with other reference ID
  - behaves like call by value

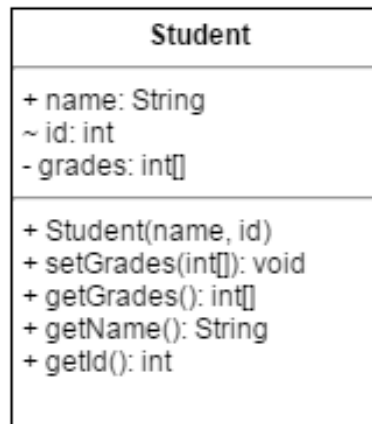
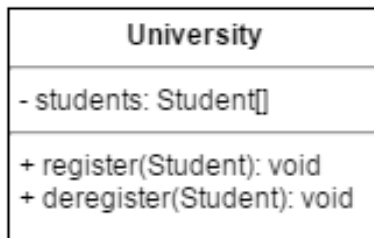
# Method call

## Multiple return values

- Is there a way to have multiple values returned by method?
  - Yes: Create a class that holds all return data and return an instance

# 1. Breakout – Exercise Suggestions (30 min.)

## 1. Model following classes and instantiate them



### Legend:

-: private access modifier  
 ~: protected access modifier  
 +: public access modifier  
*italic*: abstract

## 2.

Class *Student*, *University*:

- Are the member variables directly accessible by other classes in the same package or even in other packages?
  - Create a sub-package ,*other*' and test access to member variables of class *Student*

**Hint:** Use Eclipse Helpers: Source/Generate Getters and Setters

# 1. Breakout – Exercise Suggestions (cont.)

## 2. Answer [Questions about OOP](#)

# Object Oriented Programming

## Inheritance

# Inheritance

```
public class Dude {  
    protected String name;  
    Dude(String name) {  
        this.name=name;  
    }  
    public void sayName() {  
        System.out.println(name);  
    }  
}
```

# Inheritance

- What about special characters?
- Let's add a wizard ...
  
- But how?



# Inheritance

- What about special characters?
- Let's add a wizard...

```
public class Wizard extends Dude
{
    Wizard(String name) {
        super(name);
    }
}
```

Call constructor of super-class  
 In case of empty constructor  
 can be skipped

- Wizard can do and receive the same as Dude (apart from accessing its private fields) :

```
Dude frodo=new Dude(„frodo“);
Wizard gandalf = new Wizard(„gandalf“);
frodo.sayName();
gandalf.sayName();
```

# Inheritance

- But can't wizards do more?

```
public class Wizard extends Dude
{
    public void cast(Spell spell)
    {
        System.out.println(spell);
    }
}

gandalf.cast(someSpell);
frodo.cast(someSpell); //won't compile
```

# Inheritance

- What about method overriding?

```
public class Wizard extends Dude
{
    public void sayName() {
        System.out.println("Wizard " + name);
    }
}
```

```
gandalf.sayName(); // "Wizard Gandalf"
((Dude) gandalf).sayName(); // "Gandalf"
```

# Inheritance

- How does overriding work?
  - The JVM first looks up methods in the runtime class
  - If method is not implemented in class, JVM walks up the parent classes until method is found
  - If instance is cast to a parent class, method search starts at parent
- But:

```
Dude gandalf = new Wizard();
gandalf.sayName(); // "Wizard Gandalf"
```
- Parent types can also hold subclasses, JVM knows exact class at runtime 📁👤 **very useful for abstraction and hiding implementation details!!**

# Invisible superclass **Object**

- In Java, each class directly or indirectly inherits from class *Object*
- Object is a common superclass for each class and defines some methods
  - *equals(Object): Object*: Compare objects for equality (See block collections)
  - *hashCode(): int*: Return unique hashCode for object (See block collections)
  - ***toString() String***: Returns a *String* object containing the values of the objects' member variables. The method **must be overridden for custom classes.**

# Inheritance – Summary

- Allows classes to inherit functionality from other classes
- Allows data and procedural abstraction
- Decreases complexity of large software systems
- Reduces redundancy – reuse of common member variables and methods
- Unlike C++, Java does not support multiple inheritance, classes can only have one parent
  - but there are **Interfaces**

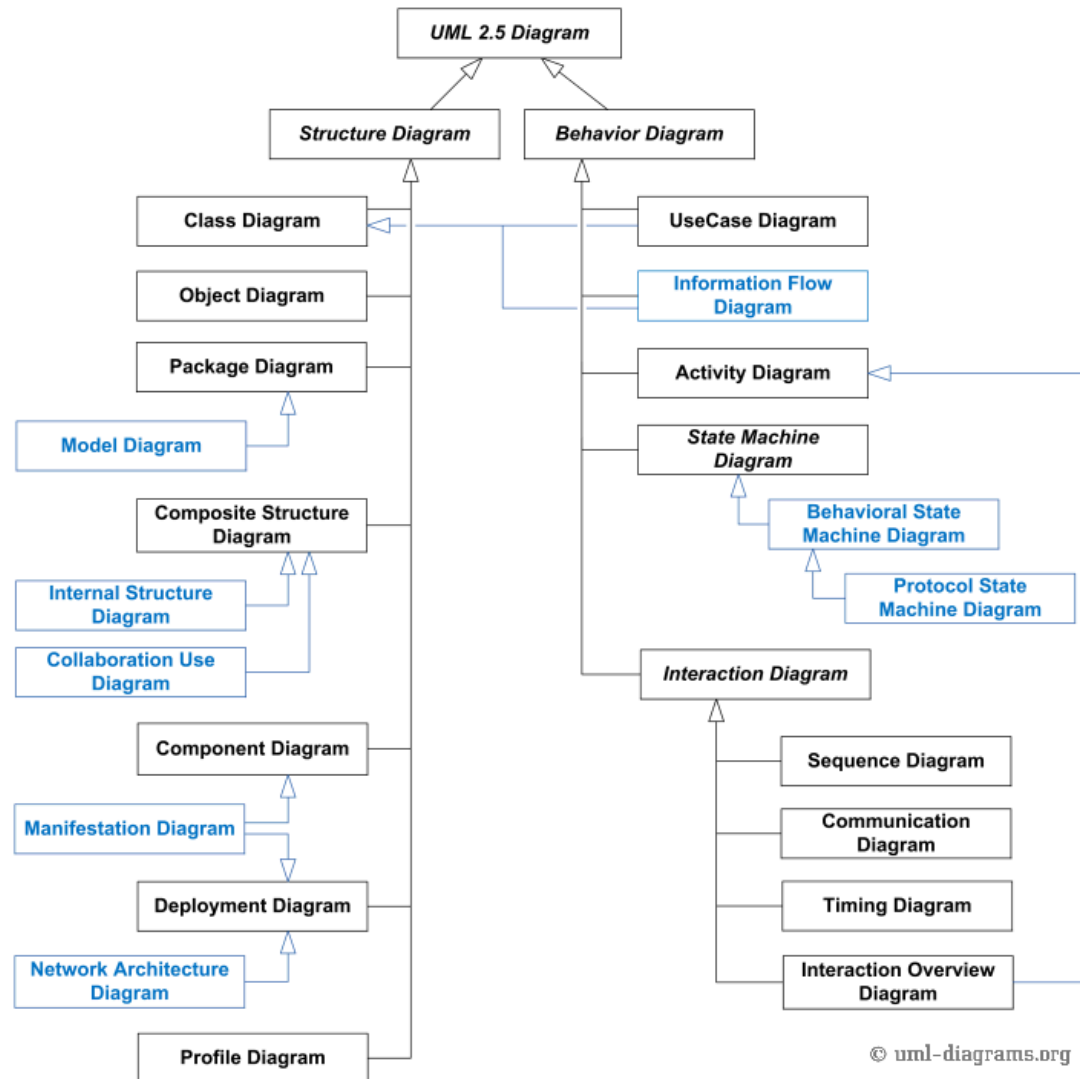
# UML

- The Unified Modeling Language™ (UML®) is a standard **visual modeling language** intended to be used for
  - modeling business and similar processes,
  - analysis, design, and implementation of software-based systems

UML is a common language for business analysts, software architects and developers used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems.

- Managed by Object Management Group (OMG)  
**Current Standard: UML v2.5, June 2015**

# UML – Overview of Diagrams





# UML – References

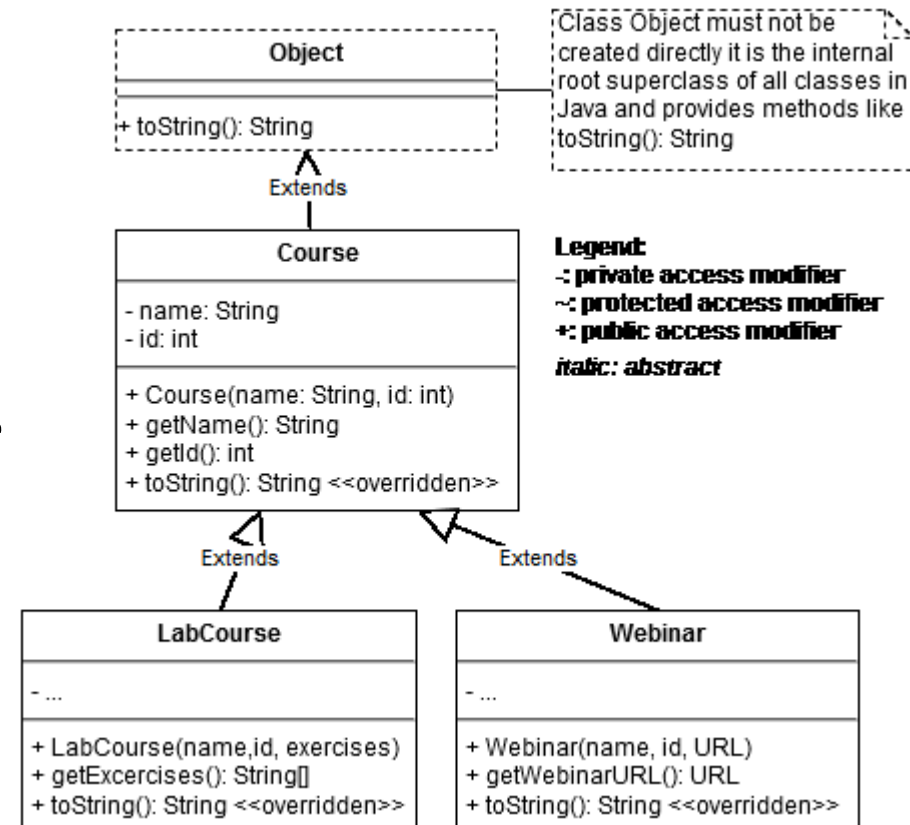
- Interesting Links
  - [UML basics](#)
  - [UML Notations Cheatsheet](#)

## Class Diagrams

## 2. Breakout – Exercise Suggestions

### 1. Model following classes and instantiate them

- Method *getName* is inherited
- Override method *toString* of class *Object*
- Create a *Course* and a *Webinar* instance and print out the object content to the console with the *toString()* method
- Print out contents of *getName()* and *getId()*



**Hint:** Use Eclipse Helpers: Source/Override/Implement Methods