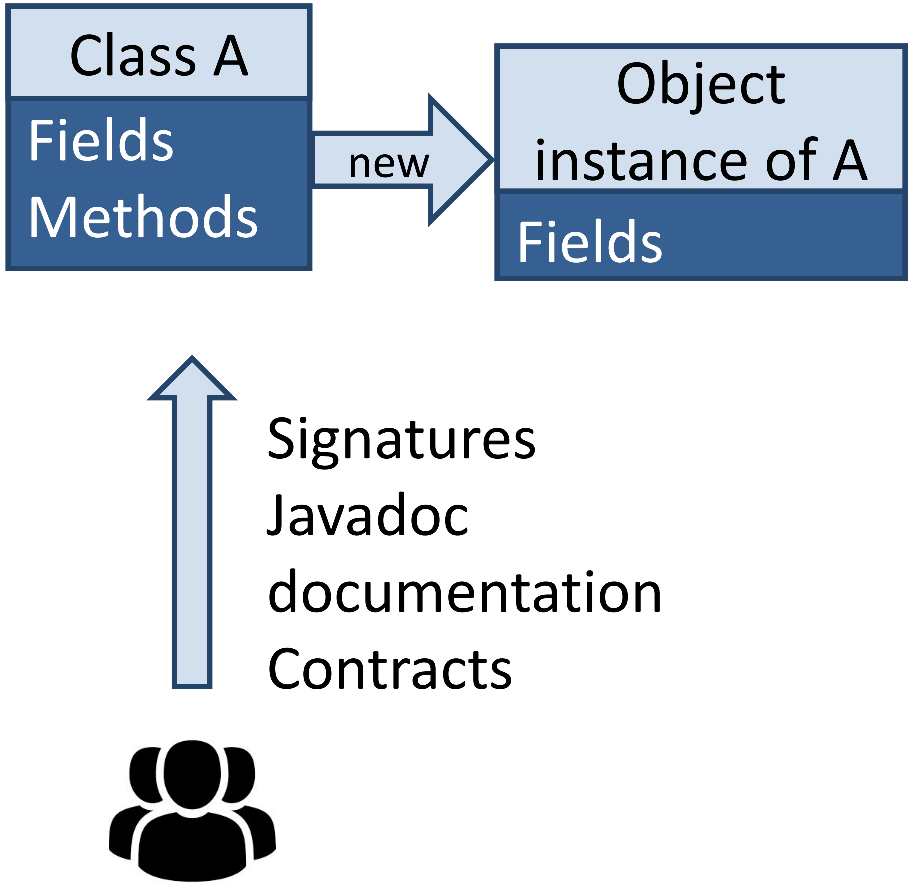
Ca' Foscari
University
of Venice

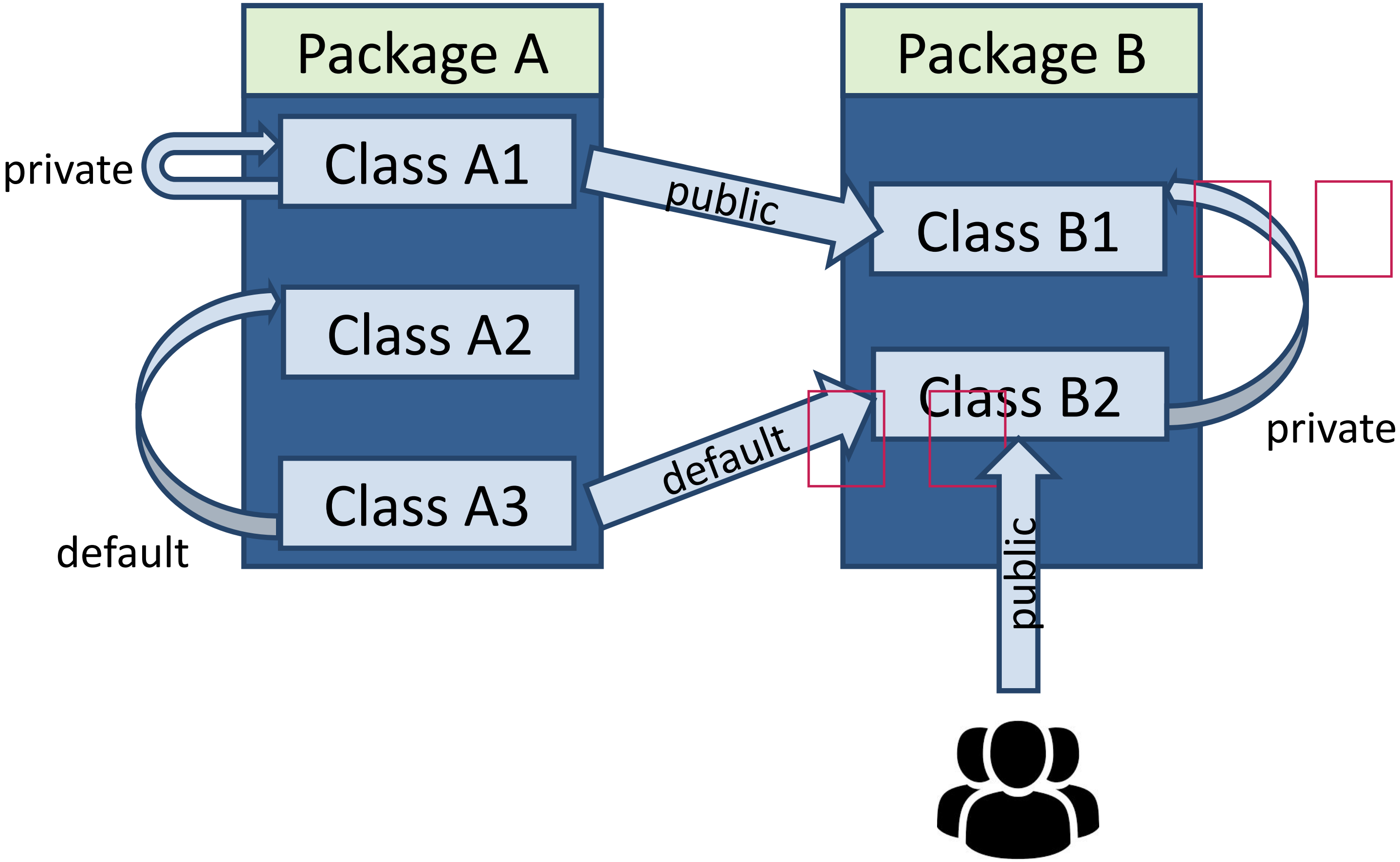
# Google DevFest

December, 2nd (Saturday)
@ Campus Via Torino
Alfa and Epsilon buildings

Sign-up here: https://devfest23.gdgvenezia.it/

# Summary


Class A
Fields
Methods
new
Object instance of A
Fields
Signatures
Javadoc
documentation
Contracts


Package A
Package B
private
Class A1
public
Class B1
Class A2
Class B2
private
default
Class A3
default
public

# Course outline

1. Encapsulation e abstraction
    a. Classi e oggetti, campi e metodi
    b. Static e final
    c. Aliasing
    d. Information hiding, attributi di visibilita’
    e. Documentazione del codice, Javadoc e file jar
    f. Java Virtual Machine e Java bytecode
    g. Design by contract

# Course outline

2. Polimorfismo
   a. Estensione di classi, overriding e overloading
   b. Abstract e final
   c. Polimorfismo, subtyping, principio di sostituzione
   d. Tipi statici e dinamici
   e. Ereditarieta' singola e multipla
   f. Classi astratte, interfacce
   g. Dispatching statico e dinamico
   h. Tipi generici

# A hierarchical view of classes

```
class Car {
  //Add the given amount to the fuel tank
  void refuel(double amount) {…}
  //Increment the speed
  void accelerate(double a) {…}
  //Stop the car
  void fullBreak() {…}
}
```

```
class Bicycle {
  //Increment the speed
  void accelerate(double a) {…}
  //Stop the car
  void fullBreak() {…}
}
```

```
class Truck {
  //Add the given amount to the fuel tank
  void refuel(double amount) {…}
  //Increment the speed
  void accelerate(double a) {…}
  //Stop the car
  void fullBreak() {…}
  //Load some stuff
  void chargeLoad(double l) {..}
  //Unload all the stuff
   double unload() {..}
}
```

Looking to the contracts:

Truck => Car
Car => Bicycle

If I need only to accelerate or break, it does not matter if I have bicycles, trucks, or cars

# Inheritance

Object oriented programming, module 1

Pietro Ferrara

pietro.ferrara@unive.it

# Aggregation

- One object refers to another one
  - has-a relation
- Car and FuelTank have a FuelType
- Two distinct objects
  - representing different concepts
- The container invokes methods of the other
- An object can aggregate many others

```
double getFuelCost() {
   return fuelType.costPerLiter;
 }
```

```
class Car {
 double speed;
 FuelType fuelType;
 double fuel;
}
class FuelType {
 String name;
 double costPerLiter;
 double fuelConsumption;
}
class FuelTank {
 FuelType type;
 double amount;
}
```

# Inheritance

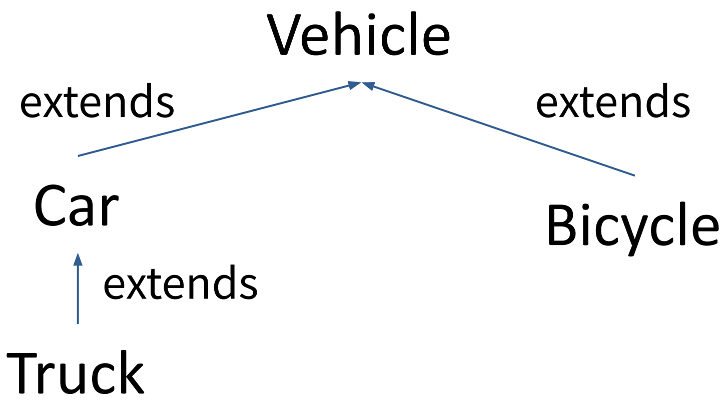
- One object extends the functionality of another
  - is-a relation
- **Reuse code** by **specializing** it
  - Add functionalities to an already defined class
- Relation fixed at compile time

```
class Vehicle {
 double speed;
 void accelerate(double a)
    {…}
}
class Car {
 double speed;
 FuelType fuelType;
 double fuel;
 void accelerate(double a)
    {…}
 void refuel(FuelTank tank)
    {…}
}
```

```
class Car extends Vehicle{
 FuelType fuelType;
 double fuel;
 void refuel(FuelTank tank) {…}
}
```

# Class extension

- Each class can extend only another one
  - Single inheritance
- A class can extend another class that extends another one
  - Inheriting all components of the super-super-class as well!
- Class hierarchy represented as a tree


Vehicle

extends

extends

Car

Bicycle

extends

Truck

```
public class Vehicle {…}
public class Car extends Vehicle {…}
public class Bicycle extends Vehicle {
  private double frontTire, rearTire;
  public double frontTirePressure() {
    return frontTire;
  }
 public double rearTirePressure() {
    return rearTire;
  }
}
public class Truck extends Car {
  private double load;
  public void chargeLoad(double l) {
    load += l;
  }
}
```

# Where did we start from?

```
typedef struct {
  int edgeLength;
} Square;

typedef struct {
  int edge1Length;
  int edge2Length;
} Rectangle;
```

```
int getAreaOfSquare(Square* s) {
  return s -> edgeLength * s -> edgeLength;
}

int getAreaOfRectangle(Rectangle * r) {
  return r -> edge1Length * r -> edge2Length;
}
```

```
int getAreaOfQuadrilateral(Quadrilateral* s) {
  switch(s -> kind) {
    case Square : return getAreaofSquare( s -> u.s);
    case Rectangle: …;
   }
}
```

```
typedef struct {
  enum { S, Re } kind;
  union  {
    Square* s;
    Rectangle* r;
   } u;
} Quadrilateral;
```

# Revised example

- A square is a specific case of a rectangle
  - is-a relationship -> subclass
- Need to properly initialize fields
- Square inherits getArea from Rectangle
- No need to define a “wrapper” class
- No duplication of code
  - Reuse getArea code of Rectangle
- Is it clear why YAPL?
  - YAPL=Yet-Another-Programming-Language

```
public class Rectangle {
 int edge1Length;
 int edge2Length;
 public int getArea () {
 return
   this.edge1Length * this.edge2Length;
 }
}

public class Square extends Rectangle {
 public Square(int edge) {
  this.edge1Length = edge;
  this.edge2Length = edge;
 }
}
```

Ca' Foscari University of Venice

# Interface of a class

- A class is composed by data (fields) and functionalities (methods)
- The interface of a class is composed by accessible methods and fields
- This changes based on where we are
  - public is visible everywhere
    - Always part of the interface
  - Default is visible only from the package
    - Usually NOT part of the interface
  - Private is visible only from the class
    - NEVER part of the interface

```
public class Vehicle {
~~private double speed;~~
 public void accelerate(double a) {
   speed += a;
 }
 public void fullBreak() {
   speed = 0.0;
 }
}
public class Car extends Vehicle {
~~private FuelType fuelType;~~
~~private double fuel;~~
 public void refuel(FuelTank tank)
     {…}
}
```

# Interface of extended classes

- Superclass: the extended class
  - Vehicle is the superclass of Car
- A class inherits all the components of the superclass
  - Class components = components defined in the class + components of the superclass

```
Car car = new Car(0, fuel, 0);
car.refuel(tank);
car.accelerate(10);
car.fullBreak();
```

```
public class Vehicle {
 ~~private double speed;~~
  public void accelerate(double a) {
    speed += a;
  }
  public void fullBreak() {
    speed = 0.0;
  }
}
public class Car extends Vehicle {
 ~~private FuelType fuelType;~~
 ~~private double fuel;~~
  public void refuel(FuelTank tank)
      {…}
}
```

# Constructors in extending classes

- Constructors of extending classes need to first call the super-constructor
  - Before anything else!
  - super(…) call
- Constructors without parameters by default
  - No constructors in both super and sub class -> constructors with no pars
  - Empty constructor of the superclass invoked by default
  - Need to explicitly invoke other super constructors with parameters

```
public class Vehicle {
 private double speed;
 public Vehicle(double speed) {
  this.speed = speed;
 }
}
public class Car extends Vehicle {
 private FuelType fuelType;
 private double fuel;
 public Car(double speed, FuelType
    fuelType, double fuel) {
  super(speed);
  this.fuelType=fuelType;
  this.fuel = fuel;
 }
}
```

# Member inheritance and access (super)

- Access components of the superclass
  - super keyword points to the instance of the superclass
  - Like this points to the current instance
- Each instance of the subclass, inherits all the components of the superclass
  - Accessible explicitly by super
- super is like this
  - Refers to the instance of the superclass

```
public class Vehicle {…}
public class Car extends Vehicle {
 private FuelType fuelType;
 private double fuel;
 public boolean isFuelEmpty() {
  if(fuel <= 0) {
   super.fullBreak();
   return true;
  }
  else
   return false;
 }
}
```

# Access modifiers

| | Same class | Same package | Subclasses | Everywhere |
|---|---|---|---|---|
| public | 👍 | 👍 | 👍 | 👍 |
| protected | 👍 | 👍 | 👍 | 👎 |
| <default> | 👍 | 👍 | 👎 | 👎 |
| private | 👍 | 👎 | 👎 | 👎 |

Covered now!

- Now we know what a subclass is!

# protected modifier

- Protected = subclasses + default
- Classes in other packages can access protected components
  - If they extend the class
- Not visible in general outside
  - Visible for who is going to extend the functionality of the class

```
public class Vehicle {
  protected double speed;
  public void fullBreak() {…}
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  public boolean isFuelEmpty() {
    if(fuel <= 0) {
      super.speed = super.speed * 0.90;
      return true;
    }
    else return false;
  }
}
```

# abstract

- Classes that implement only a part of the methods they define
  - Not implemented methods: abstract
- Cannot be instantiated
- But it can define constructors!
- Subclasses can
  - Implement all abstract methods
    - And then can be instantiated
  - Do not implement some methods
    - And then they are abstract as well

```
abstract public class Vehicle {
 protected double speed;
 abstract public void accelerate(double a);
}
public class Car extends Vehicle {
 private FuelType fuelType;
 private double fuel;
 public void accelerate(double a) {
  super.speed += a;
  this.fuel -= a * fuelType.fuelConsumption;
 }
}
public class Bicycle extends Vehicle {
 public void accelerate(double a) {
  super.speed += a;
 }
}
```

# Methods overriding

- Subclasses can override methods
  - Hide the behavior of the superclass
- super.<component> gives access to the implementation in the superclass
- Avoid to duplicate code in subclasses
  - speed+=a in accelerate
- Avoid to expose implementation details to subclasses
  - speed can be private

```
public class Vehicle {
 private double speed;
 public void accelerate(double a) {
  this.speed += a;
 }
}
public class Car extends Vehicle {
 private FuelType fuelType;
 private double fuel;
 public void accelerate(double a) {
  super.accelerate(a);
  this.fuel -= a * fuelType.fuelConsumption;
 }
}
public class Bicycle extends Vehicle {}
```

# Method definition and signature

- Method signature: everything available when calling a method
  - Type of the receiver
  - Name of the method
  - Number and static types of parameters
- Method definition: everything available when defining a method
  - Method signature and…
  - Return type
  - Visibility
  - Other modifiers (static, abstract, …)

```
public class Vehicle {
 private double speed;
 public void accelerate(double a) {
  this.speed += a;
 }
}
public class Car extends Vehicle {
 private FuelType fuelType;
 private double fuel;
 public void accelerate(double a) {
  super.accelerate(a);
  this.fuel -= a * fuelType.fuelConsumption;
 }
}
public class Bicycle extends Vehicle {}
```

Vehicle

1 double par.

Ca' Foscari University of Venice

# Overloading

- Overriding:
  - A method with exactly the same signature
  - A method hides another method in the superclass
- Overloading:
  - Several methods with the same name, different signatures, different implementations
- Do not mix up the two concepts!
  - Things will become more complex soon…

```
public class Vehicle {
  private double speed;
  public void accelerate(double a) {…}
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  public void accelerate(double a) {…}
  public void refuel(double amount) {
    fuel += amount;
  }
  public void refuel(FuelTank tank) {
    fuel += tank.getAmount();
  }
}
```

Override

Overload

# Contracts

- Method signature defines a contract
  - If a signature defines a contract less restrictive in a subclass, then it overrides
- Otherwise, it overloads
  - If the name is the same
  - And the signatures are different!
    - We must be able to distinguish at call sites
- Always think about what a class offers
- We can have a hierarchy between contracts
  - If contract A implies B, then A provides the same things of B plus something more (maybe)

```
public class Vehicle {
  private double speed;
  void accelerate(double a) {…}
}
public class Car extends Vehicle {
  public void accelerate(double a) {…}
```

Enlarge the interface
public is wider than default
Override!

```
}
public class Bicycle extends Vehicle {
  private void accelerate(double a) {…}
```

Restrict the interface
Private is stricter than default
Same signature, not allowed!

```
  void accelerate(String a) {…}
```

String is different from double
Different signature
Overload!

```
}
```

# Accessibility of overriding methods

- Modifiers are not part of the method signature
- Access modifiers can be relaxed
  - Wider visibility -> wider overriding
  - E.g., a protected method can be overridden by a public one
- Final methods cannot be overridden
- Static methods cannot be overridden
  - But we will discuss this later…

```
public class Vehicle {
  private double speed;
  protected void accelerate(double a) {
    this.speed += a;
  }
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  public void accelerate(double a) {
    super.accelerate(a);
    this.fuel -= a * fuelType.fuelConsumption;
  }
}
public class Bicycle extends Vehicle {}
```

# final methods

- How to avoid that a method is overridden?
  - final methods
  - Very different from final fields!!!
- Guarantee cannot change its behavior
  - E.g., a fullStop is a full stop!
- Other OOPL have a difference approach
  - C#: methods cannot be overridden by default
- Constructors and abstract methods cannot be final

```
public class Vehicle {
  private double speed;
 public void accelerate(double a) {
    this.speed += a;
  }
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  final public boolean accelerate(double a) {
    super.accelerate(a);
    this.fuel -= a * fuelType.fuelConsumption;
  }
}
public class Truck extends Car { … }
```

Cannot override accelerate
E.g., cheating on how much fuel is consumed

# final classes

- Prevent that a class is extended: final
- Different from marking all methods as abstract
- Protected in final class -> default
- Final classes are a serious limitation
  - Carefully evaluate if it is the case
- Better to mark all methods as final
  - Such a class can be still extended adding more methods and fields

```
public class Vehicle {
 private double speed;
 public void accelerate(double a) {
   this.speed += a;
  }
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  final public boolean accelerate(double a) {
   super.accelerate(a);
   this.fuel -= a * fuelType.fuelConsumption;
  }
}
final public class Truck extends Car { … }
```

# Types of modifiers

- ✓ Access modifiers: fields and methods
  - Only public for classes
  - Will be covered during the course
- □ □ Concurrency modifiers: fields and methods
  - Not covered
- ✓ Static: fields, methods
  - Will be covered during the course
- ✓ Final: fields, methods, classes
  - Will be covered during the course
- ✓ Abstract: methods, classes
  - Will be covered during the course

public
no modifier (default)
protected
private

**Access** ✓

synchronized
volatile

**Concurrency** □

static
final
abstract

**Others** ✓

Ca' Foscari University of Venice

# Allowed access modifiers

| | class | field | method |
|---|---|---|---|
| public | 👍 | 👍 | 👍 |
| protected | 👎 | 👍 | 👍 |
| <default> | 👍 | 👍 | 👍 |
| private | 👎 | 👍 | 👍 |

What's the benefit of giving visibility also to subclasses?

If a class is private, it cannot be referenced, extended or instantiated

# Allowed modifiers

| | class | field | method |
|---|---|---|---|
| static | 👎 | 👍 | 👍 |
| final | 👍 | 👍 | 👍 |
| abstract | 👍 | 👎 | 👍 |

For your knowledge, inner classes (not covered) can be static.

Be careful: that's different from final classes and methods!!!

Why should we allocate memory if we cannot use it?

# Allowed modifiers

| | static | final | abstract |
|---|---|---|---|
| static | | | |
| final | 👍 | | |
| abstract | 👎 | 👎 | |

But we get a warning on methods

Static: it belongs to the class. Abstract: no implementation in the class!

Abstract: no implementation. Final: we cannot extend it!

# Substituting a class with an extension

- Subclasses extend the behavior
- An instance of the superclass can be substituted by a subclass
- If we have a Vehicle, we know we can accelerate or full brake
  - No need to know if it is a bike, a car, a truck, or something else!

```
race(new Car(), new Car());
race(new Truck(), new Truck());
race(new Bicycle(), new Bicycle());
race(new Car(), new Truck());
```

```
int race(Vehicle v1, Vehicle v2, double length) {
 v1.fullStop();
 v2.fullStop();
 double distanceV1 = 0, distanceV2=0;
 while(true) {
  distanceV1 += v1.getSpeed();
  distanceV2 += v2.getSpeed();
  if(distanceV1 >= length || distanceV2 >= length) {
   if(distanceV1 > distanceV2) return 1;
   else return 2;
  }
  v1.accelerate(Math.random()*10.0);
  v2.accelerate(Math.random()*10.0);
 }
}
```

```
public class Rectangle
 extends Quadrilateral {
 int edge1Length, edge2Length;
 public Rectangle(int e1, int e2) {
  this.edge1Length = e1;
  this.edge2Length = e2;
 }
 public int getArea () { … }
}

public class Square extends Rectangle {
 public Square(int edge) {
  super(edge, edge);
 }
}
```

```
typedef struct {
 int edgeLength;
 int height;
} Rhombus;
```

```
int getAreaOfRhombus(Rhombus* r) {
 return (s -> edgeLength * s -> height(/2;
}
```

```
abstract public class Quadrilateral {
 abstract public int getArea ();
}
public class Rhombus extends Quadrilateral {
 int edgeLength, height;
 public Rhombus(int edgeLength, int height) {
  this.edgeLength = edgeLength;
  this. height = height;
 }
 public int getArea () {
  return (this.edgeLength * this.height ) / 2
 }
```

Call getArea() over a Quadrilateral and we get the area
No difference between Rectangle, Square, Rhombus, …

# Materials

- Lecture notes: Chapter 7
- Arnold & others: 3.1, 3.2, 3.3, 3.5, 3.6, 3.7, 3.10
  - Exercises: 3.1, 3.4, 3.5, 3.6
- Budd's book: 8.1, 8.2, 8.4, 8.5, 8.8, 8.9