# Object Oriented Programming in Java

**3: Classes definitions. Access modifiers. Constructors. Variable number of arguments. Static class variables and methods.**

# Licence

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- The samples and slides are inspired by the Object Oriented Programming course at the University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia.
  - Original materials in Croatian were created by (in alphabetical order): Ivica Botički, Marko Čupić, Mario Kušek, Boris Milašinović, and Krešimir Pripužić under CC-BY-NC-SA licence.
  - Adapted for this course by Boris Milašinović and shared under the same licence
  - https://creativecommons.org/licenses/by-nc-sa/4.0/

# Categorization

- Categorization: Placing things into classes or groups
- A program is developed for a (business) domain
- Each domain has
  - real life entities that must be categorized (divide in classes)
    - e.g. University: students, teachers, courses, exams, …
    - entities have attributes: name, surname, course name, grade, …
  - action related to identified categories (classes)
    - e.g. enroll, take exam, calculate grade, …
    - modelled as class methods
  - and “business” rules
    - e.g. “You have to have 90% for grade A”, …
    - rules as part of class methods

# Abstraction in object-oriented programming

- Identified objects and actions must be abstracted in order to be modelled as classes and methods
- Model only relevant attributes and methods
- What is relevant for a student?
  - name, surname, student's id are probably relevant
  - what about date of birth, advisor’s name, height, weight, …
  - it depends what we model
    - Is it a system for administrative task?
    - Is it a system for tracking sports activities?

# Encapsulation

- How TV works?
  - as long as it works and we have an interface to control it (remote control) it is not relevant
- Would students' grade would be stored inside an array or somewhere else? Do we have to know internal variable names and types?
  - allow someone to change the data directly or enforce using provided methods?

- Encapsulation
  - Bundles data with methods that operate on that data
  - Hides the implementation details and prevents unauthorized direct access

# Access modifiers in Java

- Access modifiers controls visibility and access control
  - *public*
    - Access allowed from anywhere
  - *private*
    - Accessible only inside the same class
  - *protected*
    - Accessible only to classes in the same package and to the subclasses instance (More about subclasses later in *Topic 4 – Inheritance)*
  - No modifier
    - called *package-private* visibility
    - Accessible to the classes in the same package
- Top level (e.g. classes) can be only public or package-private
  - members as variables, methods, nested classes (Topic 10) can have other modifiers

# Differences between common OOP languages

- The main difference occurs for *protected* keyword and when no access modifier is specified

| Access keyword | Java | C# | C++ |
|---|---|---|---|
| *public* | (same meaning in Java, C# and C++) | | |
| *private* | (same meaning in Java, C# and C++) | | |
| *protected* | classes from the same package and subclasses instances | subclasses instances | subclasses instances |
| *internal protected* | - | subclasses and classes from the same assembly | - |
| *private protected* | - | subclasses if they are in the same assembly | - |
| *internal* | - | classes from the same assemly | - |
| no modifier | classes from the same package | (=private) | (=private) |

# 2D Point abstraction

- A point in two-dimensional Euclidean space is represented by an ordered pair (x, y)
  - 2 class fields (attributes, variables) of *double* type
    - private fields and appropriate *getters* and *setters* to access current and set new value of encapsulated field
- Methods:
  - *print()* to write point's data to standard output
    - latter would be replaced with method *toString()*
  - *isEqualTo(Point other)* to compare a point with another one
    - Latter would be replaced with *equls(Object obj)*

# Encapsulation for Point

- Getters and setter usually named as *[get/set]VariableName*
  - *camelCase* common for Java methods

**03_Constructors/…/swu/oopj/constructors/Point.java**

```
package swu.oopj.constructors
public class Point {
    private double x, y;
    public double getX(){
        return x;
    }
    public void setX(double x){
        this.x = x;
    }
    public double getY(){
        return y;
    }
    public void setY(double y){
        this.y = y;
    }   ...
```

# Points equality (1)

- Two points are equal if they have same coordinates

**03_Constructors/…/swu/oopj/constructors/Point.java**

```
public class Point {
    ...
    public void print(){
       System.out.printf("(%.2f, %.2f)%n" , x , y);
    }

    public boolean isEqualTo(Point other) {
       return x == other.x && y == other.y;
    }
```

- Note: The solution above uses == to compare double numbers which can lead to errors due to differing precision of values
  - i.e. 3 * 0.1 is not equal to 0.3 using operator ==
  - 3 * 0.1 produces 0.3000000000000004 and 0.3 have infinite numbers of binary digits. Thus $3 * 0.1 - 0.3 \approx 5.55 * 10^{-17}$

# Points equality (2)

- A better approach is to compare absolute value of the difference with acceptable relative margin (e.g. 0.001% of one of the values)
  - for further examples $10^{-8}$ would be quite fine

**03_Constructors/…/swu/oopj/constructors/Point.java**

```
public class Point {
    private double x, y;
    ... getters and setters ...

    public void print(){
       System.out.printf("(%.2f, %.2f)%n" , x , y);
    }

    public boolean isEqualTo(Point other) {
       return Math.abs(x-other.x) < 1E-8
              &&
            Math.abs(y-other.y) < 1E-8;
    }
```

# How to create an object of Point type?

- New point (object) could be created using operator *new*

```
Point p = new Point();
```

  - In that case values for *x* and *y* would be 0 (default value for *double*)
  - Can be changed with setters *p.setX(new_value)* and *p.setY(value)*
- Can we assign some other values to x in y? Yes, by
  - providing initial value for a field in its declaration, e.g.

```
private int x = 5;
```

  - or/and by writing one or more constructors and setting variable to a value in a constructor

  - If both ways are used, variable first get the value from declaration, a then change its value to a value set in constructor

# Constructor

- Special method used to prepare new object for use (i.e. to initialize member variables to the specific values)
  - The method name is same as the class name
  - Can have arguments, but does not have return type (not even void)
  - Cannot be directly invoked
    - Executed after the memory is allocated with operator *new*
- A class can have zero or more explicitly written constructors
  - If no constructors are written by a programmer, Java compiler creates a default one with zero parameters

# A constructor for *Point*

- Constructor with two arguments: numbers that should be used as values for *x* and *y*.

```
public class Point {
  private double x, y;
  public Point(double newX, double newY){
        x = newX;
        y = newY;
  }
  ...
}
```

- A new point is created like

```
Point p = new Point(2.0, 5);
```

  - The following code is not correct anymore as there is no parameterless constructor

```
Point p = new Point();
```

    - Java compiler did not create default one because we explicitly wrote a constructor
    - We can write another one without arguments (if we want)

# Variable hiding and *this* keyword

- What if an argument name is the same as class field name?
  - E.g. what if we change *newX* and *newY* to *x* and *y* in previous constructor
  - Common in constructors and setters
- Variable name refer to the argument (and hides class variable)
- *this* is a reference to current object
  - used for example to get reference to class variable

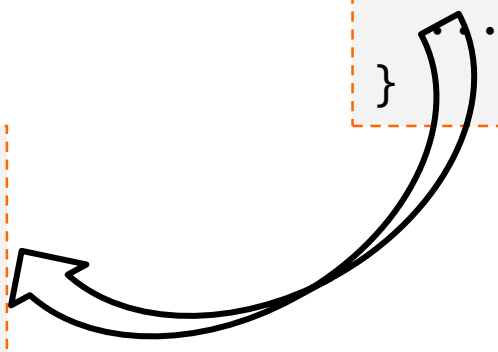
```
public class Point {
  private double x, y;
  public Point(double x, double y){
        this.x = x;
        this.y = y;
  }
  ...
}
```

# *this* to call another constructor

- Additional constructor to initialize a point based on another point
  - Repeats (almost) same code
- More elegant solution using *this*
  - Run the code from another constructor (calls an existing constructor)
  - If used, *this* must be the first statement in a constructor

```
public class Point {
  private double x, y;
  public Point(Point p) {
        x = p.x;
        y = p.y;
  }
  ...
}
```

```
public class Point {
  private double x, y;
  public Point(Point p) {
        this(p.x, p.y);
  }
  ...
}
```



Note: *this* does not create new object

- New objects are created using operator *new*

# Example using different constructors

- What is the output of the following program?

**03_Constructors/…/swu/oopj/constructors/Point.java**

```
package swu.oopj.constructors;
public class Main {
   public static void main(String[] args) {
       Point p1 = new Point(2, 5);
       Point p2 = new Point(p1);
       System.out.println("p1.isEqualTo(p2) : "
           + p1.isEqualTo(p2)); //true or false?
       p1.setX(1);
       p1.setY(2);
       System.out.println("p1.isEqualTo(p2) : "
           + p1.isEqualTo(p2)); //true or false?

       p1.print();
       p2.print();
     ...
```

```
(1.0, 2.0)   or   (1.0, 2.0)   or   (2.0, 5.0)
(2.0, 5.0)        (1.0, 2.0)        (2.0, 5.0)
```

# Static methods

- Methods *print* and *isEqualTo* are instance methods
  - In order to invoke an instance method, an object must exist
    - invoked as `object.method(arguments)`
    - Uses object's data (x and y in this case) and other methods
- Methods could be marked as static.
  - Does not require an object to be invoked
    - does not belongs to particular object, but to a class
    - invoked as `ClassName.method(arguments)`
    - cannot use non-static fields and non-static methods of the class
  - Note: Java allows calling static methods using object of the class `object.method(arguments)`
    but it should not be practiced
    - makes no sense, and e.g. not allowed in C#

# An example of static method

- Create a new point as a focus of three existing points referenced by variables *a, b, c*
- Suppose that we create this method as instance method
  - This would lead to method calls like *a.centerOf(b, c)* or some permutation of that call
  - Does not make sense because method is not intended to be part of an object, but to belong to all (three) objects, i.e. to belong to the class
    - Similarly *Integer.parseInt("12")* does not required that any integer exists before parsing the string
- Thus the method would be marked as static and called like

  *Point.centerOf(a, b, c)*

# Static method for the focus of three points (1)

- Method *centerOf* is marked as *static*
  - It creates a new point

**03_Constructors/…/swu/oopj/staticmethods/Point.java**

```
package swu.oopj.staticmethods;
public class Point
  ...
  public static Point centerOf(Point a, Point b, Point c) {
       double x = (a.x + b.x + c.x) / 3.;
       double y = (a.y + b.y + c.y) / 3.;
       Point p = new Point(x, y);
       return p;
  }
  ...
```

# Static method for the focus of three points (2)

- Method *centerOf* is static method in *Point*
- Method *print* is instance method in *Point*

**03_Constructors/…/swu/oopj/staticmethods/Main.java**

```
package swu.oopj.staticmethods;
public class Main {

      public static void main(String[] args) {
            Point a = new Point(0,0);
            Point b = new Point(6,0);
            Point c = new Point(3,5);
            Point center = Point.centerOf(a, b, c);
            center.print();
            ...
```

# Focus of multiple points (1)

- Class can have more that one methods with the same name, as long arguments name or type is different
  - The concept is called ***overloading***
  - This version receives array of points

**03_Constructors/…/swu/oopj/staticmethods/Point.java**

```
package swu.oopj.staticmethods;
public class Point
  ...
  public static Point centerOf(Point[] points){
            double x = 0, y = 0;
            int len = points.length;
            for(int i=0; i<len; i++){
                  x += points[i].x;   y += points[i].y;
            }
            Point p = new Point(x / len, y / len);
            return p;
      }  ...
```

# Focus of multiple points (2)

- Instead of classic for loop, for-each variant can be used
  - It iterates through the points array and in each pass assigns an address of next point to reference p

**03_Constructors/…/swu/oopj/staticmethods/Point.java**

```
package swu.oopj.staticmethods;
public class Point
  ...
  public static Point centerOf(Point[] points){
            double x = 0, y = 0;
            int len = points.length;
            for(Point p : points){
                  x += p.x;           y += p.y;
            }
            Point p = new Point(x / len, y / len);
            return p;
      }  ...
```

# Focus of multiple points (3)

- An array of points must be created and filled before call
- What *new Point[] {a, b, c, d}* does?
  - creates an array of 4 elements where each element is a reference to an existing point

**03_Constructors/…/swu/oopj/staticmethods/Main.java**

```
package swu.oopj.staticmethods;
public class Main {

      public static void main(String[] args) {
            ...
            Point d = new Point(7, 3);
            Point[] points = new Point[] {a, b, c, d};
            center = Point.centerOf(points);
            center.print();
            ...
```

# Focus of variables number of points (1)

- Previous solution can be used with any array size, but it is somehow inconvenient
  - Wouldn't be better to be able to call method just like in the example below?

**03_Constructors/…/swu/oopj/staticmethods/Main.java**

```
package swu.oopj.staticmethods;
public class Main {
  public static void main(String[] args) {
      Point a = new Point(0,0);
      ...
      Point.centerOf(a, b).print();
      Point.centerOf(a, b, c).print();
      Point.centerOf(a, b, c, d).print();
      Point.centerOf(a, b, c, d, new Point(4,8)).print();
            ...
```

# Focus of variables number of points (2)

- Methods can have variable number of arguments by using *Type… variable* (only) as last argument
  - Internally stored as an array

**03_Constructors/…/swu/oopj/staticmethods/Point.java**

```
public class Point
  …
  public static Point centerOf(Point a, Point b, Point...points)
  {
      double x = a.x + b.x;
      double y = a.y + b.y;
      for(Point p : points){
            x += p.x;           y += p.y;
      }
      int len = points.length + 2;
      Point p = new Point(x / len, y / len);
      return p;
  } ...
```

# Focus of variables number of points (3)

- What happens where there are more choices, e.g.

  *public static Point centerOf(Point a, Point b, Point... points)*

  *public static Point centerOf(Point a, Point b, Point c)*

  - Compiler will (if it is possible) prefer specific one to the method with variable number of arguments

```
package swu.oopj.staticmethods;
public class Main {

  public static void main(String[] args) {
      ...
      Point.centerOf(a, b).print();
      Point.centerOf(a, b, c).print();
      Point.centerOf(a, b, c, d).print();
      ...
```

**03_Constructors/…/swu/oopj/staticmethods/Main.java**

# Using *Point* in another classes

- Vector in 2D could be defined using origin and a point
  - Point to be stored inside *Vector* can be create based on two double values, or based on an existing point

**03_Constructors/…/swu/oopj/staticmethods/Vector.java**

```
package swu.oopj.staticmethods;
public class Vector {
	private Point p;
	public Vector(Point p){
		this.p = new Point(p);
	}
	public Vector(double x, double y){
		this.p = new Point(x, y);
	}
	public void print() {
		p.print();
	}
}
```

# Reference or a copy (1)?

*"The [devil / beauty / thing] is in the details"*

- What would happen if we change

```
public class Vector {
	private Point p;
	public Vector(Point p){
		this.p = new Point(p);
	}
```

to this?

```
public class Vector {
	private Point p;
	public Vector(Point p){
		this.p = p;
	}
```

# Reference or a copy (2)?

- Try to change the code and run the following excerpt

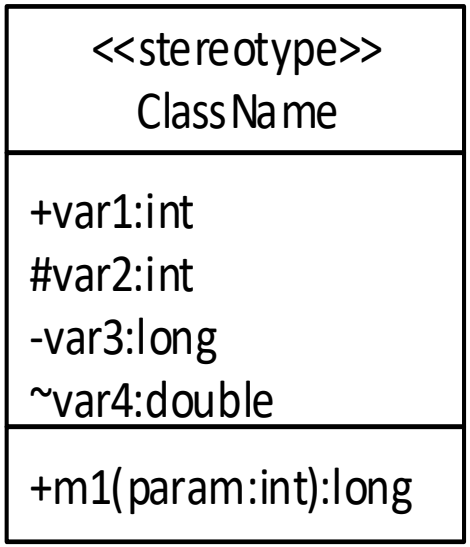
```
Point d = new Point(7, 3);
Vector v = new Vector(d);
v.print();
d.setX(17); d.setY(13);
v.print();
```

**03_Constructors/…/swu/oopj/staticmethods/Main.java**

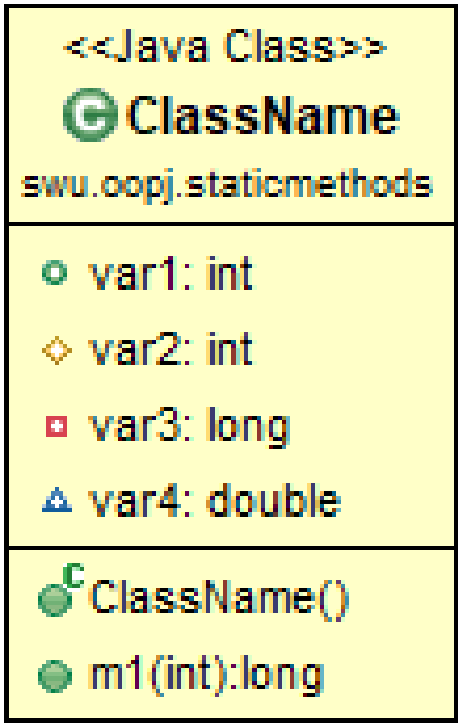
- The answer to the question depends on the problem
  - in our case, copy is more appropriate
  - Does not have to be case in future (e.g. lists, and collections in general keeps references)

# UML class diagrams

- UML = Unified Modeling Language
- Class diagrams is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

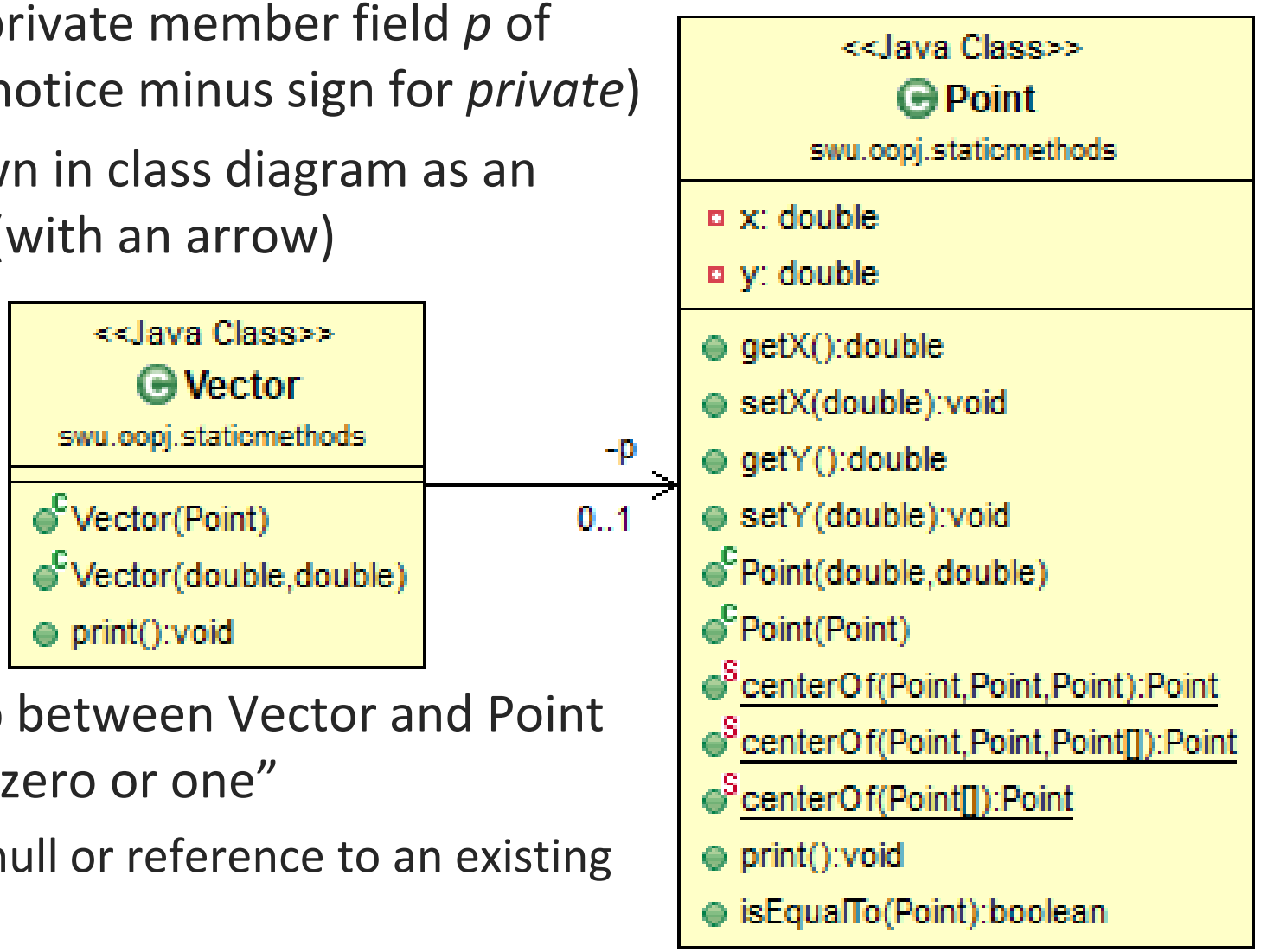



**+ public**
**# protected**
**- private**
**~ package-private**



# UML class diagram for *Vector* and *Point*

- Vector has private member field *p* of Point type (notice minus sign for *private*)
- Can be shown in class diagram as an association (with an arrow)




- Relationship between Vector and Point is “one” to “zero or one”
  - *p* can be null or reference to an existing point

# Static variables

- Belong to a class
  - Available without existence of an object of the class
- Using syntax like for static methods

  `ClassName.variableName`
- Some notable examples:
  - Math.PI, Math.E
- Usually used for constants but (as will be shown) does not have to be

# Keyword *final*

- Variables marked with *final* cannot change their value

```
final int x = 7;
...
x = 5;
```

```
final Point p = new Point(2.5, 3.0);
...
p = new Point(7.0, 4.2);
```

- However, it can change object on which refers!

```
final Point p = new Point(2.5, 3.0);
...
p.setX(7.0); p.setY(3.0)
```

- Could be static
- Final variables are initialized when declared or in constructor
  - Constant for class, or constant for an object
- Note: *final* is also used for stopping inheritance and overriding (more about that in slides T4 and T5)

# Static variables for vector space basis

- Canonical basis for $\mathbb{R}^2$ e1=(1,0) and e2=(0,1).
  - α1=(1,1) and α2=(-1, 2) is also basis in $\mathbb{R}^2$
- Each vector in $\mathbb{R}^2$ is linear combination of basis vectors.

- Should be same for all vectors => make *static*
- Cannot change canonical basic => make *final static*
  - Note: Setting final for e1 and e2 means that references are constant (see previous slide). However, as *Vector* does not provide getter for *Point*, canonical base could not be changed in the program

# Static variables initialization

- Initialization on declaration (e1, e2) or using static blocks
  - Note: C# have static constructors instead static blocks
  - Order of initialization (if both used) – on declaration then static blocks
- Static block is run before the first variable use or before first object of type Vector is created

```
package swu.oopj.staticblocks;
public class Vector {
   public final static Vector e1 = new Vector(new Point(1,0));
   public final static Vector e2 = new Vector(new Point(0,1));

   public static Vector alpha1, alpha2;
   static {
              alpha1 = e1; alpha2 = e2;
       }
  ...
```

**03_Constructors/…/swu/oopj/staticblocks/Vector.java**

# An example of using static variables

- Method print uses *EquationSolver* class to find linear combination
  - Implementation details are not relevant for the course

```
package swu.oopj.staticblocks;
public class Vector {
  ...
  public void print() {
    System.out.format("(%.2f, %.2f) = %.2f * (%.2f, %.2f) + %.2f
* (%.2f, %.2f)", ... //details are not relevant!
```

**03_Constructors/…/swu/oopj/staticblocks/Vector.java**

```
public class Main {
  Vector v = new Vector(new Point(3,4));
  v.print();
  Vector.alpha1 = new Vector(1,1);
  Vector.alpha2 = new Vector(-1,2);
  v.print();
```

**03_Constructors/…/swu/oopj/staticblocks/Main.java**

```
(3,00, 4,00) = 3,00 * (1,00, 0,00) + 4,00 * (0,00, 1,00)
(3,00, 4,00) = 3,33 * (1,00, 1,00) + 0,33 * (-1,00, 2,00)
```

# Class diagram of updated *Vector* and *Point*

- Associations removed for the sake clarity