Universidad
Carlos III de Madrid
www.uc3m.es

Lesson 6

Introduction to Object-Oriented Programming

Programming

Grade in Computer Engineering

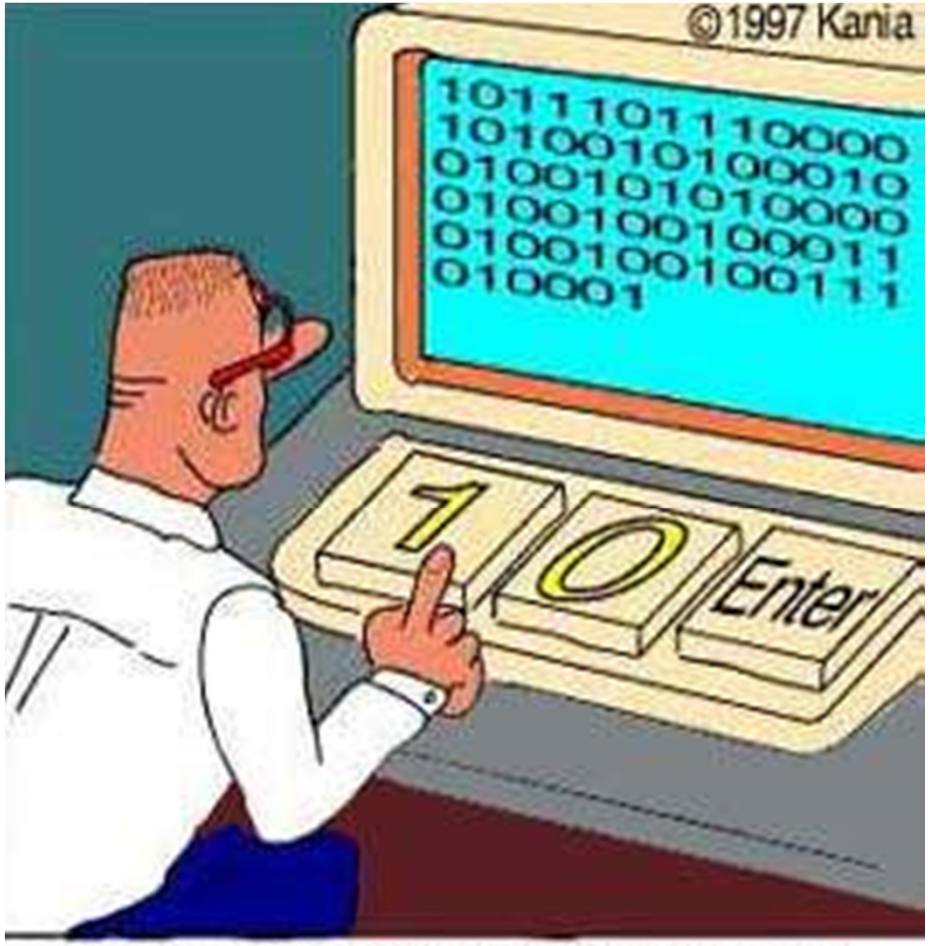
Outline

1. Motivation
2. Classes, objects and attributes
3. Constructors
4. Methods
5. Composition
6. Object destruction

Outline

1. **Motivation**
2. Classes, objects and attributes
3. Constructors
4. Methods
5. Composition
6. Object destruction

1. Motivation



Real programmers code in binary.

Object-oriented programming

Programming paradigm that defines a program as a **set of objects that perform actions** on demand

An object is **an abstract entity** of the application domain

A class defines a **blueprint for a kind of objects**

An object has

- properties <-- **Attributes!**
- operations <-- **Methods!**

which are defined in the class

Object-oriented programming is aimed **to increase system scalability** while **reducing undesired interactions** between components

Advantages

Closer to human way of thinking (more abstract)

> Improved code quality and readability

> Programs are easier to maintain

> Data and functionalities are encapsulated into meaningful structures

> Rapid development and component reuse

> Reusability and encapsulation

Outline

1. Motivation
2. **Classes, objects and attributes**
3. Constructors
4. Methods
5. Composition
6. Object destruction

An **object** can be seen as a **data structure** that represents an entity of the domain

Object Entry *#5* of an address book
- "Juan"
- "Gomez Romero"
- 29
- "jgromero@inf.uc3m.es"

Object 2D point *p*
- (2.1, 3.2)

> ***collection of values of different types*** which are managed together

An object belongs to a **class**, where the **attributes** or fields of the objects of the class are defined

Class Entry of an address book
- > Name
- > Surname
- > Age
- > E-mail

Class 2D point
- > x coordinate
- > y coordinate

Lesson 2

Programmers can define classes their own classes and use their applications

2. Classes, objects and attributes

Classes

```
[modifiers] class <name of the class> {
    <attributes>
    <methods>
}
```

[modifiers]

public	**The class can be used by any other class**
abstract	Objects cannot be created for this class, but subclasses are allowed
final	Subclasses are not allowed
none	By default, the class can be used by the classes of the same package

<name of the class>

valid Java identifier

E.g.:

```
public class Point2D {
    // stuff
}
```

2. Classes, objects and attributes

Attributes

`[modifiers]` **`<type>`** *`<name of the attribute>;`*

`[modifiers]`

`public/private``/protected`

The attribute can/cannot be directly accessed from any other class.
If protected is specified, then the attribute can be accessed from the subclasses of this class

Control attribute access and avoid inappropriate use (encapsulation!). Usually, all attributes are `private`

If none specified, the default is **`package`**, which means that the attribute can be accessed from any other class inside the package

`static`	Class attribute: it is shared by all the objects of the class
`final`	Constant (only one assignment is allowed)

E.g.:

```
public class Point2D {
    private double x;
    private double y;
}
```

Create and use class objects

Create a class with a `main` method

The program begins here

> additional methods can be created

Inside the **`main`** (or another method),

1. **Declare object variables**

   Object references are declared, but not initialized yet

2. **Create objects**

   Allocate memory for an object (create an instance of the object)

3. **Operate with objects**

   Each object stores its self values for the attributes

2. Classes, objects and attributes

Example

geometry.basic

Point2D.java | TestPoint2D_Basic.java

```
package geometry.basic;

public class Point2D {
    public double x;
    public double y;
}
```

class definition

Point2D.java | TestPoint2D_Basic.java

```
package geometry.basic;

public class TestPoint2D_Basic {
    public static void main(String [] args) {
        Point2D p1 = new Point2D();
        p1.x = 1.0;
        p1.y = 1.0;
        System.out.println("Point 1 at (" + p1.x + ", " + p1.y + ")");

        Point2D p2 = new Point2D();
        p2.x = 3.0;
        p2.y = 4.0;
        System.out.println("Point 2 at (" + p2.x + ", " + p2.y + ")");


        double distance;
        distance = Math.sqrt( Math.pow(p1.x-p2.x, 2) + Math.pow(p1.y-p2.y, 2) );
        System.out.println("Distance: " + distance);
    }

}
```

public attributes
reading and writing object attributes

TestPoint2D_Methods.java

```
public class TestPoint2D_Methods {

    public static void readCoordinates(Point2D p) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Reading coordinates: ");
        System.out.println("x? ");
        p.x = sc.nextDouble();
        System.out.println("y? ");
        p.y = sc.nextDouble();
    }

    public static void printCoordinates(Point2D p) {
        System.out.println("(" + p.x + ", " + p.y + ")");
    }

    public static double distance(Point2D a, Point2D b) {
        double distance;
        distance = Math.sqrt( Math.pow(a.x-b.x, 2) + Math.pow(a.y-b.y, 2) );
        return distance;
    }

    public static Point2D midPoint(Point2D a, Point2D b) {
        Point2D mid = new Point2D();
        mid.x = (a.x + b.x) / 2.0;
        mid.y = (a.y + b.y) / 2.0;
        return mid;
    }

    public static void main(String [] args) {
        Point2D p1 = new Point2D();
        readCoordinates(p1);
        System.out.print("Point 1 at ");
        printCoordinates(p1);

        Point2D p2 = new Point2D();
        readCoordinates(p2);
        System.out.print("Point 2 at ");
        printCoordinates(p2);

        System.out.println("Distance: " + distance(p1, p2));

        Point2D m = midPoint(p1, p2);
        System.out.print("Midpoint at ");
        printCoordinates(m);
    }
}
```

public attributes
reading and writing object attributes from any method

objects as parameters
objects can be used as method parameters

objects as returning values
objects can be used as method returning values

using objects in the program

geometry.methods

Object arrays can be also declared, but memory for the array and for the objects must be allocated

Array allocation

```
Point2D [] points;
points = new Point2D[5];        // 5-elements array
```

Array elements allocation

```
points[0] = new Point2D();      // point at pos. 0
points[1] = new Point2D();
```

Using array elements

```
points[0].x = 1.0;      // x coordinate of point at pos. 0
points[0].y = 2.0;      // y coordinate of point at pos. 0
```

If an array position is not allocated, the default value is *null*. Thus, if we try to access to its attributes, we get a **runtime exception**

```
points[3].x = 1.0;      // Runtime error, points[3] is null
```

Create a Java program based on *TestPoint2D_Methods* class to read 10 points from the keyboard and calculate the two closest points

geometry.collections

An object can represent a **collection of values**

Matrix2D

Mathematical notion of matrix: collection of values which can be accessed with two indexes (i, j)

Attributes

- 2D matrices are represented with a two dimensional array of doubles
- 2D matrices have a size (number of rows, number of columns)

myMatrix1

elements

1.0	2.0	3.4	1.1	-1.1
-1.2	1.0	2.9	2.3	0.3
7.4	4.4	1.1	0.0	1.0
-3.1	2.2	-2.1	-0.2	0.0

rows: 4 cols: 5

myMatrix2

elements

3.3	2.1	0.7	-1.9
12.2	2.1	9.3	0.01
4.9	1.1	7.9	-0.1

rows: 3 cols: 4

Arrays can be used as attributes in class definitions

Array attributes are accessed by using **.** (to access the attribute) and **[]** (to access the array element)

Matrix.java | TestMatrix_Basic.java

```
public class Matrix {
    public double [][] elements;
    public int rows;
    public int cols;
}
```

matrix.basic

Matrix.java | TestMatrix_Basic.java

```
package matrix.basic;

public class TestMatrix_Basic {

    public static void main(String [] args) {

        Matrix m = new Matrix();
        m.rows = 3;
        m.cols = 4;
        m.elements = new double[3][4];

        for(int i=0; i<m.rows; i++) {
            for(int j=0; j<m.cols; j++) {
                m.elements[i][j] = Math.random() * 10;
                System.out.printf("%.2f ", m.elements[i][j]);
            }
            System.out.println();
        }
    }
}
```

Notice the difference between:

an array of objects

```
Point2D [] points;
points = new Point2D[5];
System.out.println(points[1].x);
System.out.println(points[1].y);
```

an object with an array as attribute

```
Matrix m;
m = new Matrix();
m.elements = new double[3][4];
System.out.println(m.elements[2][3];
```

Combination is possible: we can have an array of objects that include arrays as attributes

an array of matrix objects

```
Matrix [] matrices;
matrices = new Matrix[3];
matrices[0] = new Matrix();
matrices[0].elements = new double[4][5];
matrices[0].elements[2][3] = 12.1;
```

2. Classes, objects and attributes

Arrays of objects vs objects with array attributes

Matrix.java | TestMatrix_Basic.java | TestMatrix_Collections.java

```
package matrix.basic;

public class TestMatrix_Collections {

    public static void main(String [] args) {

        Matrix [] matrices = new Matrix[5];

        for(int k=0; k<matrices.length; k++) {
            matrices[k] = new Matrix();     // allocating matrix objects in the array

            matrices[k].rows = 2;           // initializing matrix object attributes
            matrices[k].cols = 3;
            matrices[k].elements = new double[2][3];    // initialize internal array!

            System.out.println("\nmatrix " + k);

            for(int i=0; i<matrices[k].rows; i++) {
                for(int j=0; j<matrices[k].cols; j++) {
                    matrices[k].elements[i][j] = Math.random() * 10;
                    System.out.printf("%.2f ", matrices[k].elements[i][j]);
                }
                System.out.println();
            }
        }
    }

}
```

Static vs. non-static attributes

Non-static (or instance) attributes are attributes local to an object instance

- Each object of a class has its own values
  - E.g.: coordinates (x, y) of Point2D
- To access non-static attributes, an object instance must be created

Static (or class) attributes are attributes shared by all the objects of a class

- `static` modifier is used
- Common values for all the objects of a class
  - E.g.: constants, counters, etc.
- To access static attributes, it is not necessary to create an object of the class
- static attributes are automatically initialized with default values if no initial value is provided –although usually values are assigned in the class definition
- Frequently, static attributes are `final` (they cannot be changed)

2. Classes, objects and attributes

Example

student

Student.java | TestStudent.java

```
package student;

public class Student {
    public String name;
    public String surname;
    public int age;

    public double mark1stPartialExam;
    public double mark2ndPartialExam;
    public double mark1stPracticalExercise;
    public double mark2ndPracticalExercise;
    public double mark3rdPracticalExercise;
    public double markJanuaryExam;
    public double markJuneExam;

    public static final String university = "University Carlos III of Madrid";
    public static int nStudentsRegistered;
}
```

Static attributes definition

Static attributes are defined with the **`static`** modifier

2. Classes, objects and attributes

Example

Student.java | TestStudent.java

student

```
public class TestStudent {
    public static void main(String [] args) {

        System.out.println("Registered " +
                Student.nStudentsRegistered + " at " +
                Student.university);

        Student st1 = new Student();
        Student.nStudentsRegistered++;

        /* Creating first student */
        st1.name = "John";
        st1.surname = "Doe";
        st1.mark1stPartialExam = 5.5;
        st1.mark2ndPartialExam = 6.3;
        st1.mark1stPracticalExercise = 8.5;
        st1.mark2ndPracticalExercise = 7.2;
        st1.mark3rdPracticalExercise = 7.7;
        st1.markJuneExam = 6.8;

        double mark1 =
            st1.mark1stPartialExam * 0.1 +
            st1.mark2ndPartialExam * 0.1 +
            st1.mark1stPracticalExercise * 0.05 +
            st1.mark2ndPracticalExercise * 0.1  +
            st1.mark3rdPracticalExercise * 0.15  +
            st1.markJuneExam * 0.4;

        System.out.println(st1.name + " " + st1.surname + ": " + mark1);

        /* Print information */
        System.out.println("Registered " +
                Student.nStudentsRegistered + " at " +
                Student.university);
```

Static attributes access

Static attributes are accessed with the **name of the class**, dot (.) and the name of the attribute
No objects of the class are created to access the static attribute

Non-static attributes access

2. Classes, objects and attributes

Example

Student.java | TestStudent.java

```
        /* Creating second student */
        Student st2 = new Student();
        Student.nStudentsRegistered++;

        st2.name = "Jane";
        st2.surname = "Eod";
        st2.mark1stPartialExam = 7.5;
        st2.mark2ndPartialExam = 6.8;
        st2.mark1stPracticalExercise = 5.8;
        st2.mark2ndPracticalExercise = 7.1;
        st2.mark3rdPracticalExercise = 9.95;
        st2.markJuneExam = 8.3;

        double mark2 =
                st2.mark1stPartialExam * 0.1 +
                st2.mark2ndPartialExam * 0.1 +
                st2.mark1stPracticalExercise * 0.05 +
                st2.mark2ndPracticalExercise * 0.1  +
                st2.mark3rdPracticalExercise * 0.15  +
                st2.markJuneExam * 0.4;


        System.out.println(st2.name + " " + st2.surname + ": " + mark2);

        /* Print information */
        System.out.println("Registered " +
                Student.nStudentsRegistered + " at " +
                Student.university);
    }
}
```

student

Console

```
<terminated> TestStudent [Java Application] C:\Program Files (x86)\
Registered 0 at University Carlos III of Madrid
John Doe: 6.200000000000001
Registered 1 at University Carlos III of Madrid
Jane Eod: 7.242500000000001
Registered 2 at University Carlos III of Madrid
```

Outline

1. Motivation
2. Classes, objects and attributes
3. **Constructors**
4. Methods
5. Composition
6. Object destruction

Point2D.java | TestPoint2D_Basic.java

```
package geometry.basic;

public class TestPoint2D_Basic {
    public static void main(String [] args) {
        Point2D p1 = new Point2D();
        p1.x = 1.0;
        p1.y = 1.0;
        System.out.println("Point 1 at (" + p1.x + ", " + p1.y + ")");

        Point2D p2 = new Point2D();
        p2.x = 3.0;
        p2.y = 4.0;
        System.out.println("Point 2 at (" + p2.x + ", " + p2.y + ")");


        double distance;
        distance = Math.sqrt( Math.pow(p1.x-p2.x, 2) + Math.pow(p1.y-p2.y, 2) );
        System.out.println("Distance: " + distance);
    }

}
```

Attribute value assignment

Non-static attributes are assigned right after the allocation of the object

It would be convenient to **provide an initial value for the attributes of a new object**

Usually, after creating an object, some initializations may be convenient

The coordinates of a Point object must be initialized
The internal array of a Matrix must be allocated

A special method is created inside the class template to initialize object attributes: **constructor**

Constructors perform all the initializations required to create a valid object of a class

Constructors are executed when calling to **`new`**

Memory is allocated for the object, then the constructor is executed

The syntax of the call to **`new`** must correspond to one of the constructors of the class; otherwise, we get a compilation error

3. Constructors

Syntax

```
[modifiers] <class name> (
      [ <type of parameter 1> <name of parameter 1>  [,
        <type of parameter 2> <name of parameter 2> …] ] ) {
         <sentences>
}
```

[modifiers]

public/private/protected

E.g.:

```
public class Point2D {
   public Point2D(double x_value, double y_value) {
      …
   }
}
```

3. Constructors

Syntax

Point2D.java | TestPoint2D_Constructor.java

```
public class Point2D {
    public double x;
    public double y;

    public Point2D(double x_value, double y_value) {
        x = x_value;
        y = y_value;
    }
}
```

geometry.constructor

Point2D.java | TestPoint2D_Constructor.java

```
public class TestPoint2D_Constructor {
    public static void main(String [] args) {
        Point2D p1 = new Point2D(1.0, 1.0);
        System.out.println("Point 1 at (" + p1.x + ", " + p1.y + ")");

        Point2D p2 = new Point2D(3.0, 4.0);
        System.out.println("Point 2 at (" + p2.x + ", " + p2.y + ")");

        double distance;
        distance = Math.sqrt( Math.pow(p1.x-p2.x, 2) + Math.pow(p1.y-p2.y, 2) );
        System.out.println("Distance: " + distance);
    }

}
```

Constructors are **never** static

Constructors do not return any value (**they do not have returning type!**)

The instructions inside the constructor can access to the object attributes. These attributes correspond to the object instance currently that is being initialized

More than one constructor could be implemented (with different parameter number or type)

If not implemented, a default constructor with no arguments and empty code is assumed. In this case, the fields of the object are assigned default values (0: numbers and characters, false: booleans, null: Strings, arrays, objects)

If there is at least one constructor with parameters, the default constructor is no longer valid

matrix.constructor

Constructors use

Matrix.java | TestMatrix_Basic.java

```
public class TestMatrix_Basic {

    public static void main(String [] args) {

        System.out.println("\nmatrix 1");
        Matrix m1 = new Matrix(3, 4);
        for(int i=0; i<m1.rows; i++) {
            for(int j=0; j<m1.cols; j++)
                System.out.printf("%.2f ", m1.elements[i][j]);
            System.out.println();
        }

        System.out.println("\nmatrix 2");
        Matrix m2 = new Matrix(3, 4, 0, 2);
        for(int i=0; i<m2.rows; i++) {
            for(int j=0; j<m2.cols; j++)
                System.out.printf("%.2f ", m2.elements[i][j]);
            System.out.println();
        }

        // Matrix m3 = new Matrix();    // Compilation error!
    }
}
```

Matrix.java | TestMatrix_Basic.java

```
public class Matrix {
    public double [][] elements;
    public int rows;
    public int cols;

    public Matrix(int nRows, int nCols) {
        rows = nRows;
        cols = nCols;
        elements = new double[rows][cols];
    }

    public Matrix(int nRows, int nCols, double a, double b) {
        rows = nRows;
        cols = nCols;
        elements = new double[rows][cols];

        for(int i=0; i<rows; i++)
            for(int j=0; j<cols; j++)
                elements[i][j] = a + (b-a)*Math.random();
    }

}
```

Constructors definition

Outline

1. Motivation
2. Classes, objects and attributes
3. Constructors
4. **Methods**
5. Composition
6. Object destruction

Objects as **data structures + operations**

An object can be seen as a data structure that represents an entity of the domain + **A set of related operations applied on the object**

Matrix

Attributes

3 rows

2 columns

Values $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$

Operations

get values

print

add (to other matrix)

subtract (to other matrix)

...

4. Methods
Example

Class Matrix2D

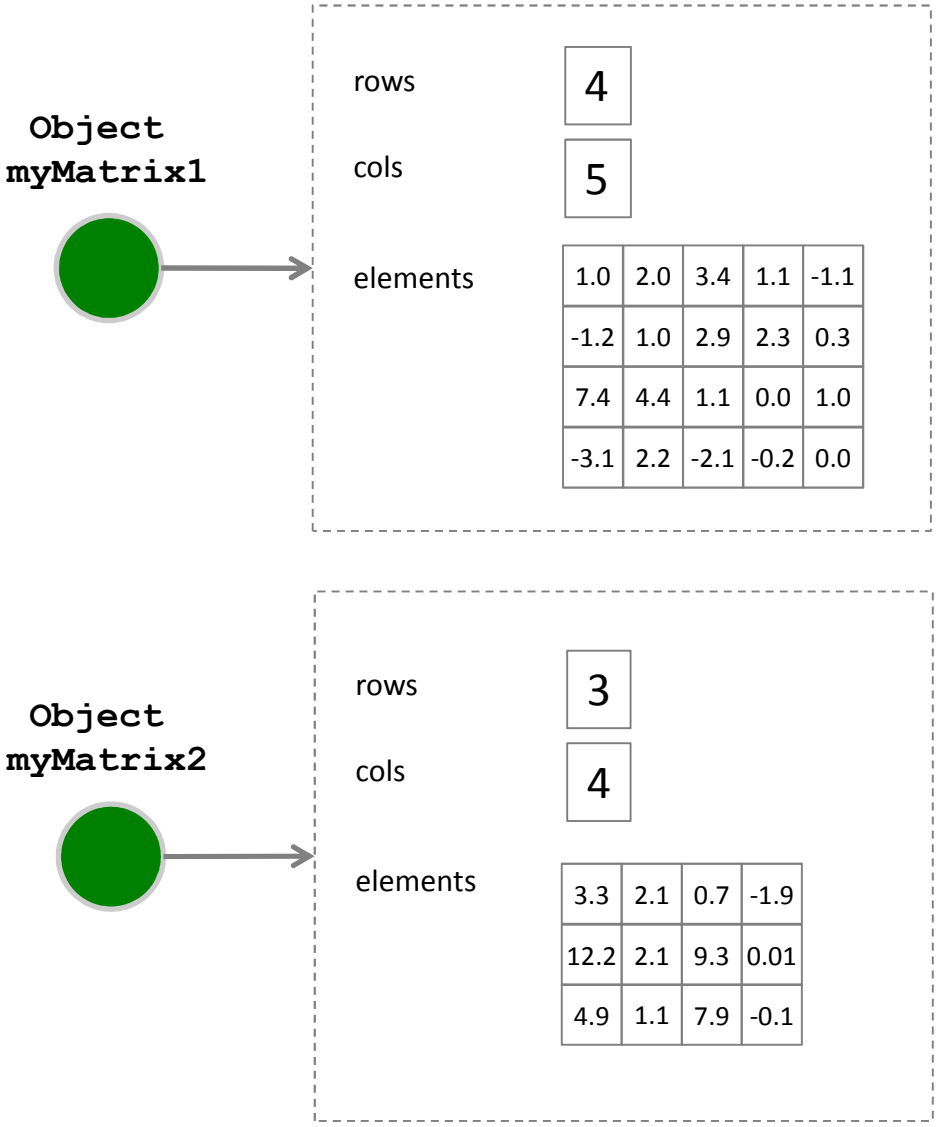
- elements (double [][])
- number of rows (int)
- number of columns (int)

- retrieve element at position (i, j) (double)
- assign element *x* at position (i, j)
- add to other Matrix2D (Matrix2D)

What is your **number of columns**, *myMatrix1*?

What is your **value at position (1, 3)**, *myMatrix2*?

Add myMatrix2 to you, *myMatrix1*



Methods are included in the class definition

Static methods (also named class methods)

- Methods previously studied
- General methods
- Return values calculated from the parameter values
- Cannot access to non-static attributes & can access to static attributes
- `static` keyword

Non-static methods (also named instance methods)

- Methods applied on an object of the class
- Modify values of the object instance
- Return values calculated from the object instance values and the parameters
- Can access to non-static attributes & can access to static attributes
- Without the `static` keyword

Non-static method use

Methods are applied on an object of the class (which can be considered as their implicit parameter)

Methods are called by using the dot (.) operator

```
myMatrix1.getRows()
myMatrix1.add(myMatrix2)
```

Non-static method definition

Methods have access to attributes defined in the class. The value of the attributes is retained through calls

Methods can define local variables. Their visibility extends over the code appearing within the same scope ({ … })

Methods can be overloaded: we can define two methods with the same name and different number of parameters

Matrix (reloaded)

> Create Matrix class

> Attributes

- rows
- cols
- values

> Constructor

- allocate memory for *values*

> Methods

- read values from keyboard
- random initialization
- print
- get value at position (i, j)
- set value at position (i, j)
- add to other matrix
- get transpose
- etc.

Recommendations

Use **`public`** for the class

Use **`private`** for all the attributes, instead of **`public`** –they cannot be accessed from any other class

Define at least one **`public`** constructor

Define proper get/set methods –we can control how object attributes are modified

Use **`public`** for the methods –unless they are internal methods

The accessibility to attributes and methods depends on **scope modifiers**:

`public` The entity can be used from any package

An attribute can be accessed from methods of other class
A method can be invoked from methods of other class

`private` The entity cannot be accessed from *outside the class*

An attribute can be directly accessed only from the methods of the class; it is not accessible even from subclasses (same for methods)

`protected` The entity can be only accessed from *inside* the subclass

An attribute can be only accessed from the methods of the subclasses (same for methods)

`none` The entity can be accessed only from the package

An attribute can be only accessed from methods in classes of the same package (same for methods)

The general scope rule is valid for local variables: a variable is accessible only from the block in which it has been declared

4. Methods
Example

matrix.methods

Matrix.java

```
public class Matrix {

    private int rows;
    private int cols;
    private double [][] values;

    /** Constructor */
    public Matrix(int r, int c) {
        rows = r;
        cols = c;
        values = new double[r][c];
    }

}
```

Matrix.java | TestMatrix.java

```
public class TestMatrix {

    public static void main(String[] args) {

        Matrix m;
        m = new Matrix(3, 4);

    }

}
```

4. Methods

Example – accesibility modifiers

Matrix.java

```
public class Matrix {

    private int rows;
    private int cols;
    private double [][] values;

    /** Constructor */
    public Matrix(int r, int c) {
        rows = r;
        cols = c;
        values = new double[r][c];
    }

}
```

Matrix.java | TestMatrix.java

```
public class TestMatrix {

    public static void main(String[] args) {

        Matrix m;
        m = new Matrix(3, 4);

        System.out.println(m.rows);

    }

}
```

Compilation error:
The field *Matrix.rows* is not accesible (not visible)

4. Methods

Example – method definition

Matrix.java | TestMatrix.java

```
public class Matrix {

    private int rows;
    private int cols;
    private double [][] values;

    /** Constructors */
    public Matrix(int r, int c) {
        rows = r;
        cols = c;
        values = new double[r][c];
    }

    /** Random initialization of the matrix */
    public void randomInitialization() {

        // Initialize values
        for(int i=0; i<rows; i++)
            for(int j=0; j<cols; j++)
                values[i][j] = Math.random();
    }

    /** Print matrix */
    public void print() {
        for(int i=0; i<rows; i++) {
            for(int j=0; j<cols; j++)
                System.out.print(values[i][j] + " ");
            System.out.println();
        }
    }
}
```

Example – object declaration

Matrix.java | TestMatrix.java

```
public class TestMatrix {

    public static void main(String[] args) {

        Matrix m;
        m = new Matrix(4, 5);
        m.randomInitialization();
        m.print();

    }

}
```

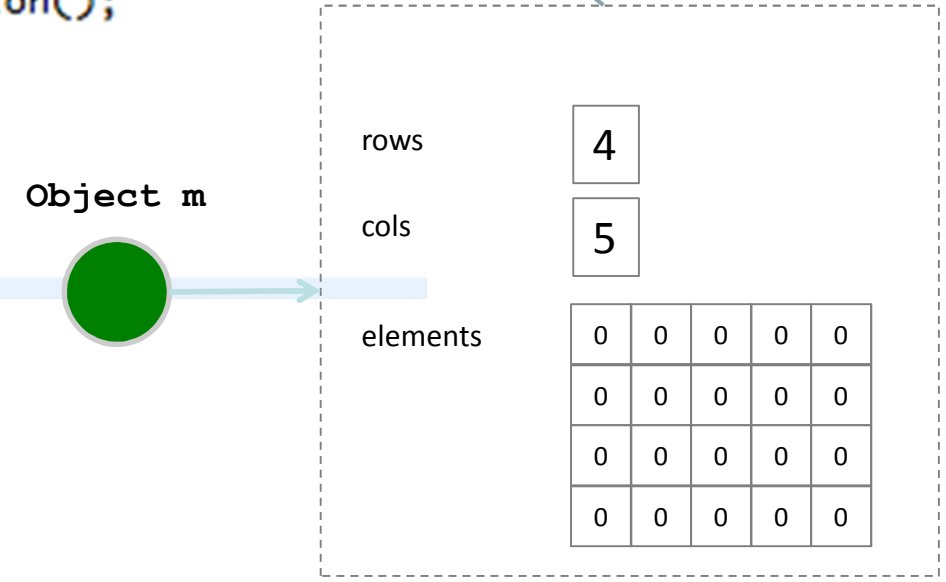
Object m

4. Methods

Example – object allocation

Matrix.java | TestMatrix.java

```
public class TestMatrix {

    public static void main(String[] args) {

        Matrix m;
        m = new Matrix(4, 5);
        m.randomInitialization();
        m.print();

    }

}
```



4. Methods

Example – method call

Matrix.java | TestMatrix.java

```
public class TestMatrix {

    public static void main(String[] args) {

        Matrix m;
        m = new Matrix(4, 5);
        m.randomInitialization();
        m.print();

    }

}
```

Object m

rows	4
cols	5

elements

0.4	0.2	0.1	0.5	0.6
0.2	0.0	0.2	0.7	0.1
0.8	0.8	0.0	0.2	0.7
0.2	0.3	0.9	0.5	0.1

Matrix.java | TestMatrix.java

```
public class TestMatrix {

    public static void main(String[] args) {

        Matrix m;
        m = new Matrix(4, 5);
        m.randomInitialization();
        m.print();

    }

}
```

Console

```
<terminated> TestMatrix [Java Application] /Syste
Rows? 4
Cols? 5
0.4282160817157613 0.24115419022720652
0.20569676864259545 0.08455189161647869
0.8362341838258047 0.8808909542511871 0
0.21648854685553687 0.3313274935314998
```

Object m

rows	4
cols	5

elements

0.4	0.2	0.1	0.5	0.6
0.2	0.0	0.2	0.7	0.1
0.8	0.8	0.0	0.2	0.7
0.2	0.3	0.9	0.5	0.1

```java
/** Add matrices
 *  @param m Matrix (m must proper size)
 *  @return Result of the addition operation */
public Matrix add(Matrix m) {
    Matrix r = new Matrix(rows, cols);
    for(int i=0; i<rows; i++)
        for(int j=0; j<cols; j++)
            r.values[i][j] = values[i][j] + m.values[i][j];

    return r;
}
```

Non-static
method definition

```java
/** Adds input matrices A and B */
public static double [][] add(double [][] A, double [][] B) {
    double [][] C = null;

    C = new double[A.length][A[0].length];

    for(int i=0; i<A.length; i++)
        for(int j=0; j<A[0].length; j++)
            C[i][j] = A[i][j] + B[i][j];


    return C;
}
```

Static
method definition

Calls to **non-static** methods

Matrix.java | TestMatrix.java

```
package matrix.methods;

public class TestMatrix {

    public static void main(String[] args) {

        Matrix m;
        m = new Matrix(4, 5);
        m.randomInitialization(0, 1);
        m.print();

        Matrix n;
        n = new Matrix(4, 5);
        n.randomInitialization(0, 1);

        Matrix r = n.add(m);
        r.print();
    }
}
```

Calls to **static** methods

Exercise6.java

```
    public static void main(String [] args) {

        double [][] m, n, r;

        m = new double[3][4];
        randomInitialization(m, 0, 1);

        n = new double[3][4];
        randomInitialization(n, 0, 1);

        r = add(m, n);
        print(r);
    }
```

```
    public static void main(String [] args) {

        double [][] m, n, r;

        m = new double[3][4];
        Exercise6.randomInitialization(m, 0, 1);

        n = new double[3][4];
        Exercise6.randomInitialization(n, 0, 1);

        r = add(m, n);
        Exercise6.print(r);
    }
```

Inside a method implementation, an object can refer to itself with **`this`** keyword

`this` is implicitly defined in the method (it is not necessary to declare or initialize it)

```
/** Constructors */
public Matrix(int r, int c) {
    this.rows = r;
    this.cols = c;
    this.values = new double[r][c];
}

/** Random initialization of the matrix */
public void randomInitialization() {

    // Initialize values
    for(int i=0; i<this.rows; i++)
        for(int j=0; j<this.cols; j++)
            this.values[i][j] = Math.random();
}
```

this must be used when disambiguation is required

A parameter can be defined with the same name as an attribute. By default, if this is not used, the parameter is used

```
public class Point {

    private double x;
    private double y;

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

    public void setX(double x) {
        this.x = x;
    }
```

this can be also used to invoke a constructor from inside other constructor

In this case, the call with this must be the first instruction of the constructor

```java
public class Point {

    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public Point() {
        this(0, 0);
    }
```

```java
public class Point {

    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public Point() {
        this.x = 0;
        this.y = 0;
    }
```

Both implementations are equivalent

Scope issues

(The same as for methods defined in Lesson 5)

Lesson 2

1. A **copy of the value** passed as argument is stored in the parameter
2. The scope of formal parameters and local variables **is the method**

The values of basic-type variables and the references to arrays and objects used as actual parameters **are not changed** inside methods

The values of arrays and the attributes of objects used as actual parameters **may be changed** inside methods

Outline

1. Motivation
2. Classes, objects and attributes
3. Constructors
4. Methods
5. **Composition**
6. Object destruction

The attributes of a class can be objects

Point.java | Triangle.java | TestPointComposition.java

```
package geometry.composition;

public class Point {

    private double x;
    private double y;

    public Point() {
        this(0, 0);
    }
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public void setX(double x) {
        this.x = x;
    }
    public void setY(double y) {
        this.y = y;
    }
    public double getX()  {
        return x;
    }
    public double getY() {
        return y;
    }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
    public double distance(Point p) {
        return Math.sqrt(Math.pow(x-p.x, 2) + Math.pow(y-p.y, 2));
    }
}
```

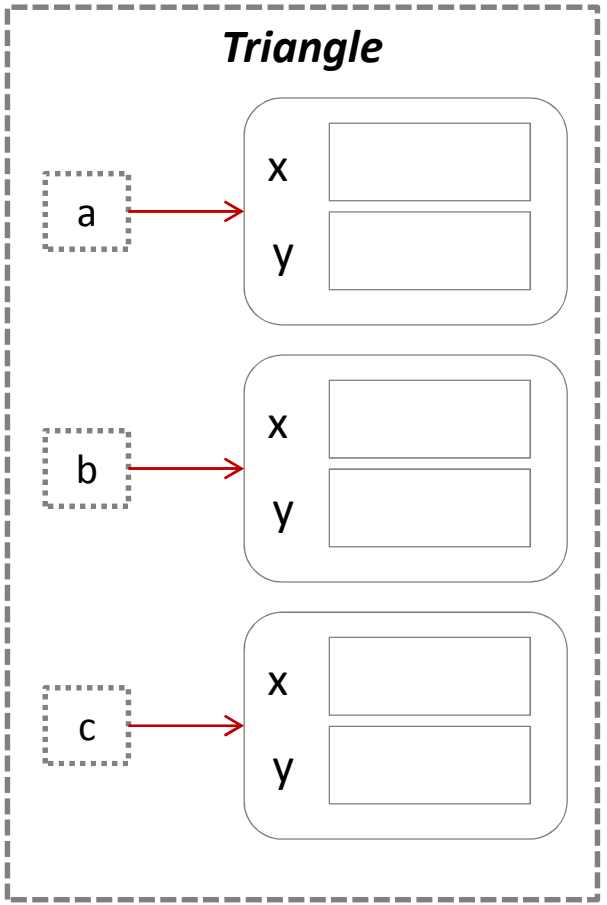
geometry.composition

5. Composition

Point.java | Triangle.java | TestPointComposition.java

```
package geometry.composition;

public class Triangle {

    private Point a;
    private Point b;
    private Point c;

    public Triangle() {
        this(new Point(), new Point(), new Point());
    }
    public Triangle(Point a, Point b, Point c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public void setA(Point a) {
        this.a = a;
    }
    public void setB(Point b) {
        this.b = b;
    }
    public void setC(Point c) {
        this.c = c;
    }
    public Point getA() {
        return a;
    }
    public Point getB() {
        return b;
    }
    public Point getC() {
        return c;
    }

    public String toString() {
        return a.toString() + ", " + b.toString() + ", " + c.toString();
    }
    public double perimeter() {
        double p = a.distance(b) + b.distance(c) + c.distance(a);
        return p;
    }

}
```

Object references



watch out object references!
every object in the composition must be properly allocated (**constructors**) to avoid null pointer exceptions

5. Composition

Example – using composite objects

Point.java | Triangle.java | TestPointComposition.java

```
package geometry.composition;

public class TestPointComposition {
    public static void main(String [] args) {

        // Triangle: (0, 0), (3, 0), (3, 2)
        Point p1 = new Point(0, 0);
        Point p2 = new Point(3, 0);
        Point p3 = new Point(3, 2);
        Triangle t = new Triangle(p1, p2, p3);
        System.out.println("coordinates: " + t.toString());
        System.out.println("perimeter:   " + t.perimeter());

        // Triangle: (-1, -1), (2, 1), (3, 3)
        Triangle r = new Triangle(new Point(-1, -1), new Point(2, 1), new Point(3, 3));
        System.out.println("coordinates: " + r.toString());
        System.out.println("a: " + r.getA().toString());
        System.out.println("b: " + r.getB().toString());
        System.out.println("c: " + r.getC().toString());

        // change point
        r.setA(new Point(-1, -2));
        System.out.println("coordinates: " + r.toString());

        // use points of r and t
        // (distance between first vertex of r and t)
        double d = r.getA().distance(t.getA());
        System.out.println("distance: " + d);
    }

}
```

Console

```
<terminated> TestPointComposition [Java Application] C:\Program Files (x86)\
coordinates: (0.0, 0.0), (3.0, 0.0), (3.0, 2.0)
perimeter:   8.60555127546399
coordinates: (-1.0, -1.0), (2.0, 1.0), (3.0, 3.0)
a: (-1.0, -1.0)
b: (2.0, 1.0)
c: (3.0, 3.0)
coordinates: (-1.0, -2.0), (2.0, 1.0), (3.0, 3.0)
distance: 2.23606797749979
```

Outline

1. Motivation
2. Classes, objects and attributes
3. Constructors
4. Methods
5. Composition
6. **Object destruction**

In Java, it is not necessary to explicitly release the memory

The garbage collector is an automatic procedure that frees the memory associated to unused object and array references

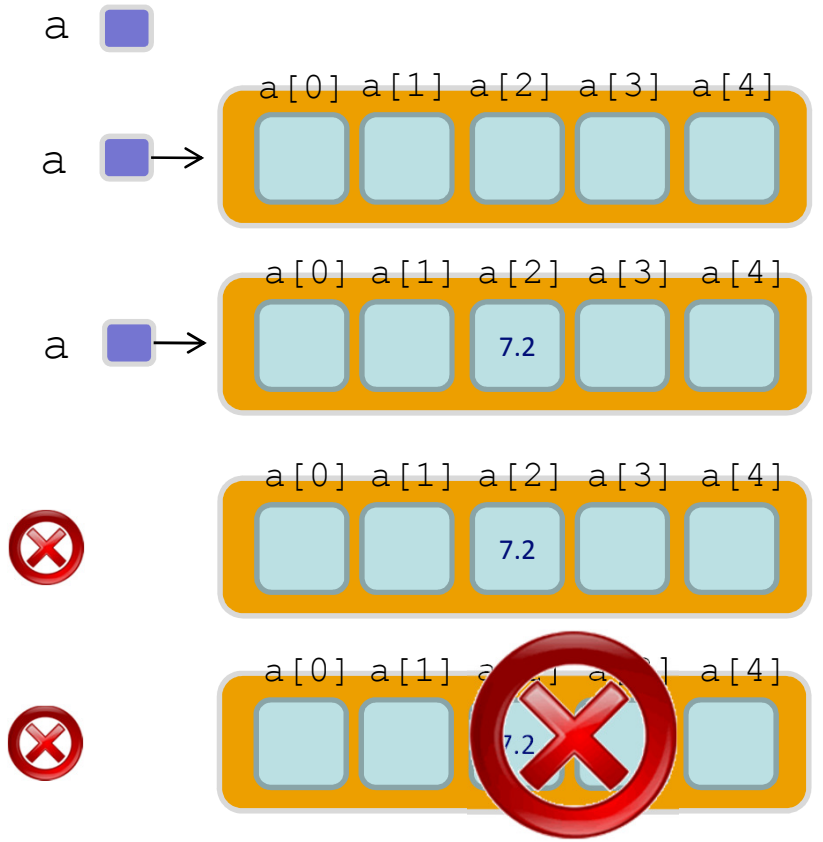
E.g.: The memory assigned to a local array is marked as free by the garbage collector when the method ends

The case for an object **is equivalent**

6. Object destruction

Example – memory released

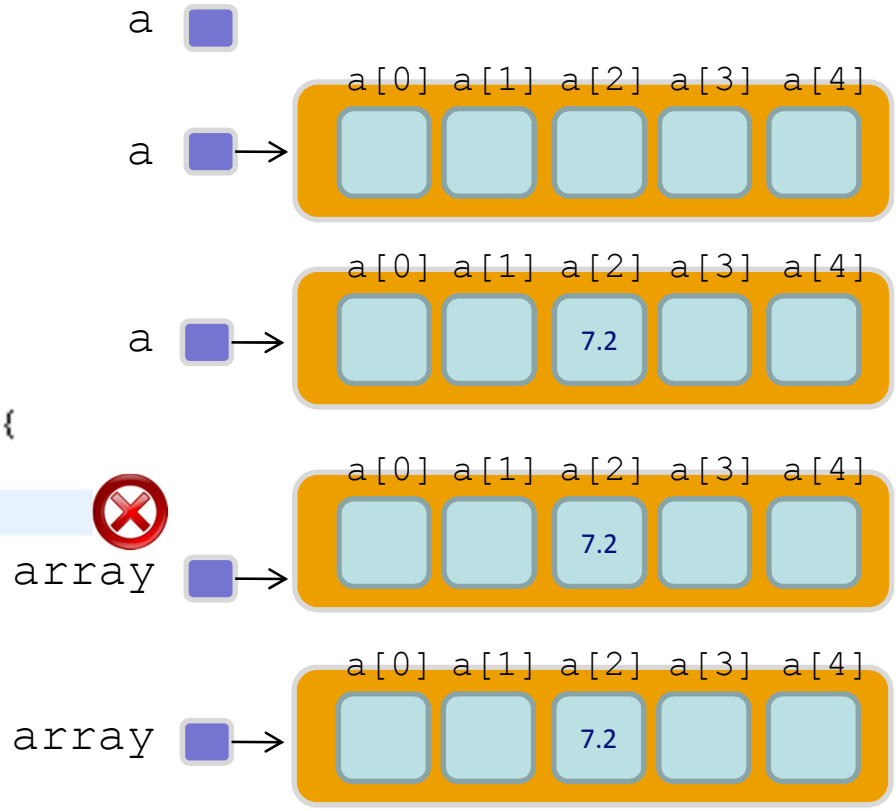
```
public void method1() {
    double [] a;
    a = new double[5];
    a[2] = 7.2;
}

public static void main(String[] args) {
    DestructionExample d =
        new DestructionExample();

    d.method1();

}
```



6. Object destruction

Example – memory is alive

```
public double [] method2() {
     double [] a;
     a = new double[5];
     a[2] = 7.2;
     return a;
}


public static void main(String[] args) {
    DestructionExample d =
         new DestructionExample();

    double [] array;
    array = d.method2();

}
```



6. Object destruction
finalize()

Code can be executed when the object is destroyed by the Garbage Collector: create (*override*) method **finalize**

E.g.: Close a file that is open while the object is alive

```
protected void finalize() {
    // Clean-up operations
    System.out.println("An object is finalized");
}
```

It can not be known exactly when the garbage collector will be triggered. If there is no lack of memory, it is probable that it will be never triggered

It is advisable not to rely on it for conducting any other task

You can explicitly call the Garbage Collector with `System.gc()`, although this is only a "suggestion" to the JVM

Outline

1. Motivation
2. Classes, objects and attributes
3. Constructors
4. Methods
5. Composition
6. Object destruction

Classes

Attributes (properties)

- private

```
private int x;
```

- arrays

```
private double [][] values;
```

- composition (objects)

```
private Point a;
```

Methods (behavior)

- constructor
  - object attribute initialization
- get/set methods
  - retrieve and change values of private attributes
- other methods
  - object functionalities

Objects

References

single reference

```
Point p;
```

arrays of objects

```
Point [] points = new Point[5];
```

Object allocation

```
p = new Point();
p = new Point(1, 2);
points[0] = new Point();
```

Method calling

```
int x_coordinate = p.getX();
int y_coordinate = points[0].getX();
```

Recommended lectures

H. M. Deitel, P. J. Deitel. *Java: How to Program. Prentice Hall, 2011 (9th Edition)*, **Chapters 6** [link]**, 8** [link]

The Java™ Tutorials. Oracle, **Classes and objects** [link]

I. Horton. *Beginning Java, Java 7 Edition*. Wrox, 2011, **Chapter 5** [link]

Programming – Grado en Ingeniería Informática	
Authors	 Universidad Carlos III de Madrid
Of this version: Juan Gómez Romero *Based on the work by:* Ángel García Olaya Manuel Pereira González Silvia de Castro García Gustavo Fernández-Baillo Cañas	