Object Oriented Programming in Java

Etienne Duris

Université Gustave Eiffel – ESIPE - IGM

Some good lectures

Java Language & Virtual Machine Specifications

https://docs.oracle.com/javase/specs/index.html

Doug Lea’s coding conventions

http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

Effective Java, 2nd/3rd Edition (Joshua Bloch)

Crowedsourced Java questions

https://stackoverflow.com/questions/tagged/java

Rémi Forax home page (and support!)

http://www-igm.univ-mlv.fr/~forax/ens/java-avance/cours/pdf/

Several programming styles

Imperative (Algol, FORTRAN, Pascal, C…)

Sequences of instructions describe (how the result is obtained by manipulating the memory state (variables)

Declarative (Prolog, SQL…)

Statements of what you get or what you want, rather than how to achieve the result

Applicative or **functional** (LISP, Caml, Haskel…)

Based on expression or function evaluations where the result does'nt rely on memory state (no *side effect*)

Object Oriented (modula, Objective-C, Self, C++...)

Reusable units to abstract interactions and control side effects

Why control / avoid side effects

A **side effect** is a memory state modification (or input/output) that imply a change in a program behavior

- Difficult to debug, since hard to reproduce
- Requires an external synchronization mechanism if several execution threads could reach a shared memory zone

When possible, **avoid** side effect

At least, try to **control** it

Object oriented programming style

Objects are autonomous components with their own resources and able to communicate with each other

These **objects** represent **data** that are modeled by **classes**; these classes define **types**

Like a `typedef struct` define a type in C

Types also define **actions** that objects can perform and how they affect their state

messages or **methods**

Benefits of object oriented programming

Abstraction

Separation between definition (what) and implementation (how)

Unification

Data and code could be unified in a single model

Reusability

Class design leads to reusable components (distinguishing and separating concepts)

Hide implementation details

Specialization (no so true in real life, actually)

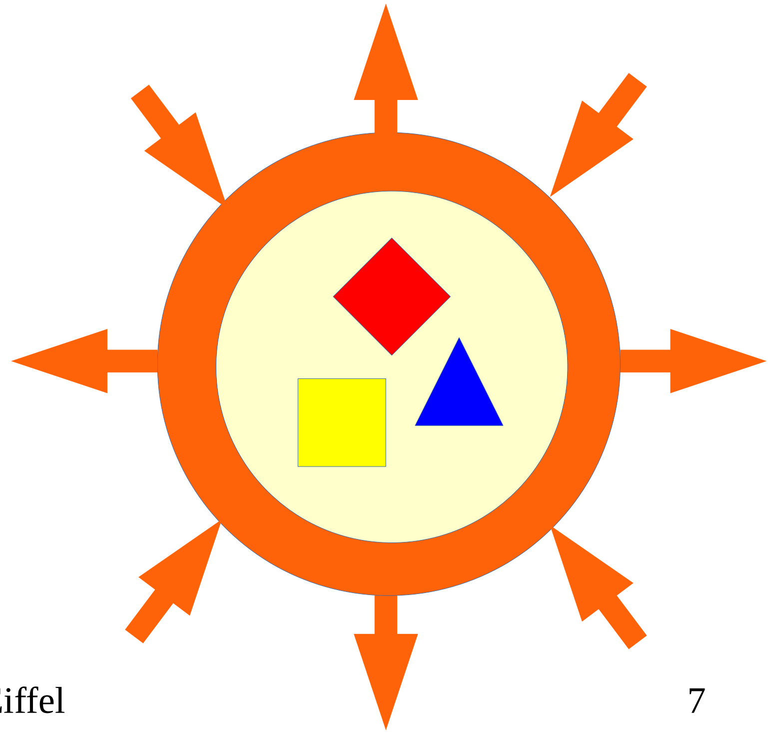
Inheritance mechanism allows specialization in specific situations

Modular programming

Class design, representing both data, actions and responsibilities of this class objects, allows programmer to **distinguish and separate concepts**

"**Interface**" definition (the way to communicate with the world outside)",
hides implementation details and avoid too strong dependencies

This promotes reusability and **composition / delegation**:
the assembly of the components with respect to their responsibilities



What is an object?

It defines **inside** and **outside**

Outside **should NOT know** how inside works.

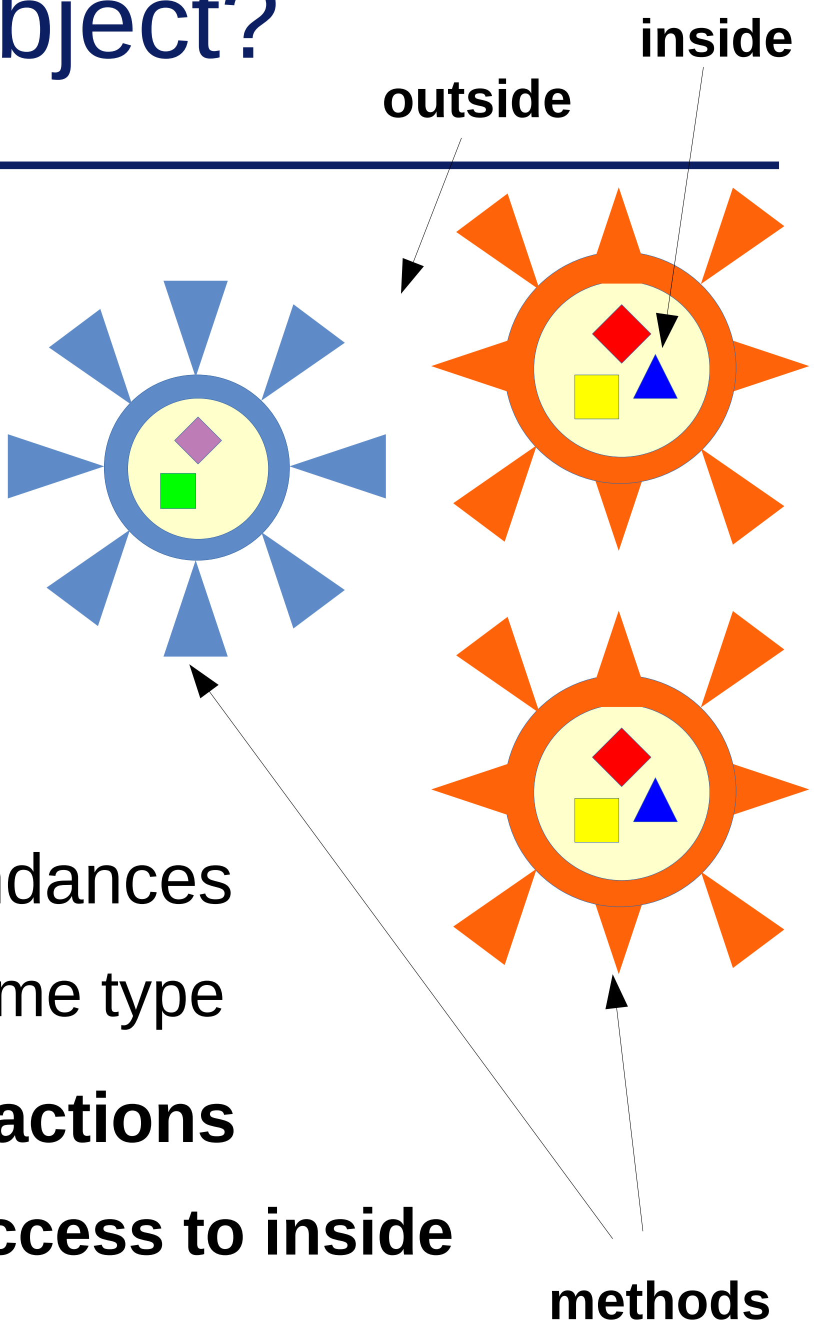
- inside = good (what is controled)
- outside = evil (what is not controled)

Forbid direct access to inside to avoid mistakes and strong dependances

- softened view between objects of same type

Instead, **use methods to perform actions**

- called from outside, **they do have access to inside**

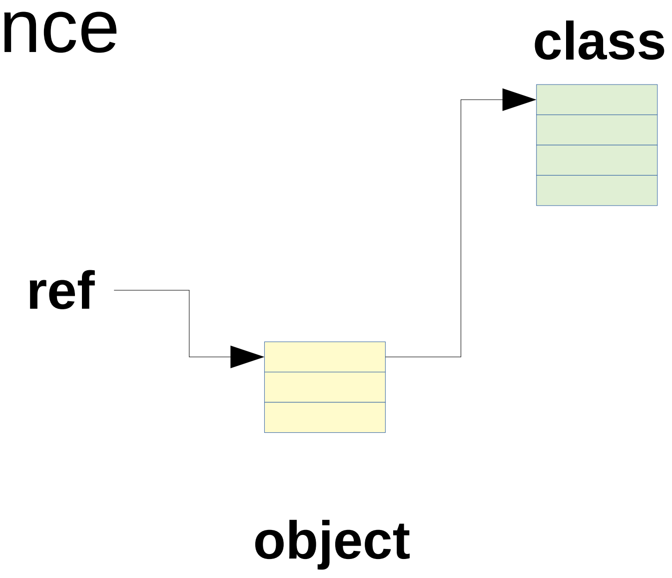


From memory point of view

An object is stored in a memory area

It is usually handled through a reference

- In Java, we do not talk about “pointer” since no arithmetic is available on references -- just access
- In Java, each object knows its size
- In Java, each object knows its class

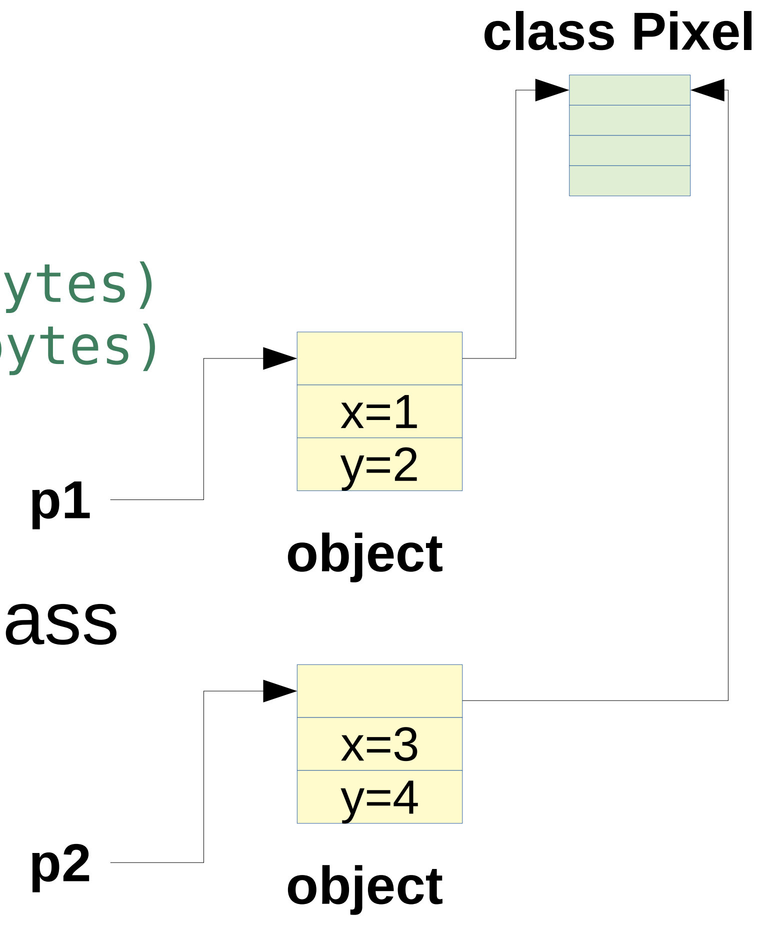


Object = instance of a class

The memory area inside an object is formatted by its class definition

Like a `struct` in C

```
class Pixel {
   int x; // signed int (32 bits, 4 bytes)
   int y; // signed int (32 bits, 4 bytes)
}
```



All objects (**instances**) of a same class are identically formatted in memory but each has **its own state** (distinct values)

Class and fields

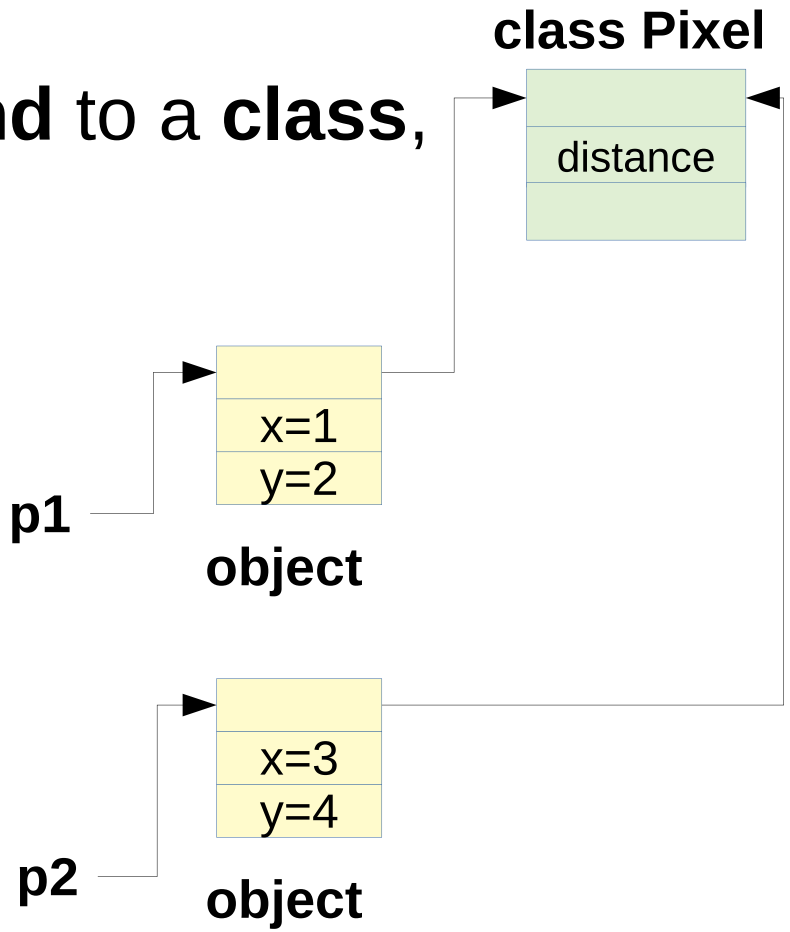
A class defines the memory structure of its objects

- Each **field** (attribute), with its **type**, implies a memory area, size and layout
- In Java, the order of the fields in memory is not necessarily the same as the order of declaration (contrary to C)
- The whole size of an object is often larger than the sum of its field’s sizes
  - due to alignment in memory
  - and due to special fields, present in each object (like a reference to its class)

Class and methods

In addition to fields, a class defines the code that deals with them, i.e. **methods**

A method is a **function** that is **bound** to a **class**, **a**nd through the class, it is bound to each object of this class




```
class Pixel {
   int x;
   int y;
   double distance() {
      return Math.sqrt(x*x + y*y);
   }
}

p1.distance(); // 2.23606797749979
p2.distance(); // 5.0
```

Methods

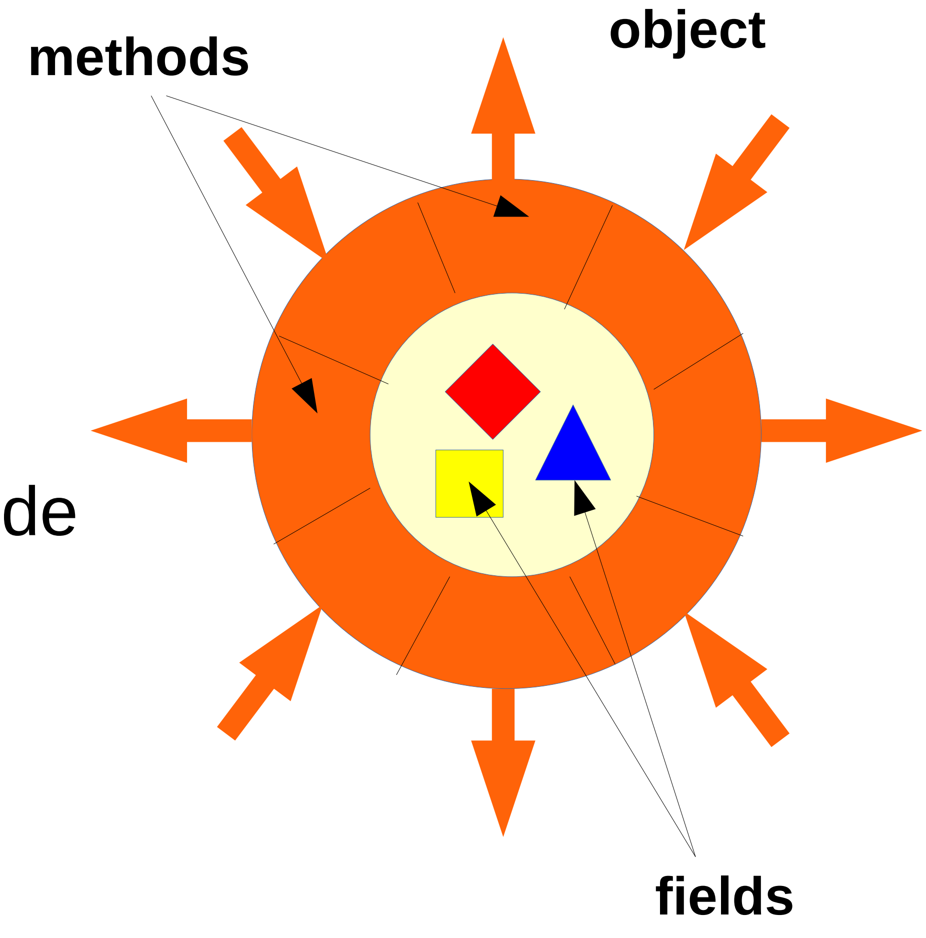
In addition to fields, a class defines the code that deals with them, i.e. **methods**

Methods allow objects to interact each others

Outside should interact with an object through its methods

- They guard inside against outside

All the fields of an object are reachable from methods of its class



Method and method call

The execution of a method is necessarily associated to an **object** (an instance) of a class: the **receiver** of the method call

We say that the method is called “on” this receiver object

```
Pixel p1 = …
p1.distance();
```

receiver

```
Scanner sc = ...
sc.nextLine();
```

When the method is executed, it has access to the values of this instance’s fields (and only **this** one)

```
double distance() {
   return Math.sqrt(this.x*this.x + this.y*this.y);
}
p1.distance(); // 2.23606797749979
p2.distance(); // 5.0
```

A method is a function with a hidden parameter

This hidden parameter stands for the receiver object reference, known as **this** in Java

```
class AnotherClass {
   void foo() {
      Pixel p = ...
      p.printXAndY();
   }
}
```

```
class Pixel {
  int x;
  int y;

  void printXAndY() {
    System.out.println(this.x + " " + this.y);
  }
}
```

When `p.printXandY()` is called, `this` refers to the value of `p` in execution of the code of `printXandY`

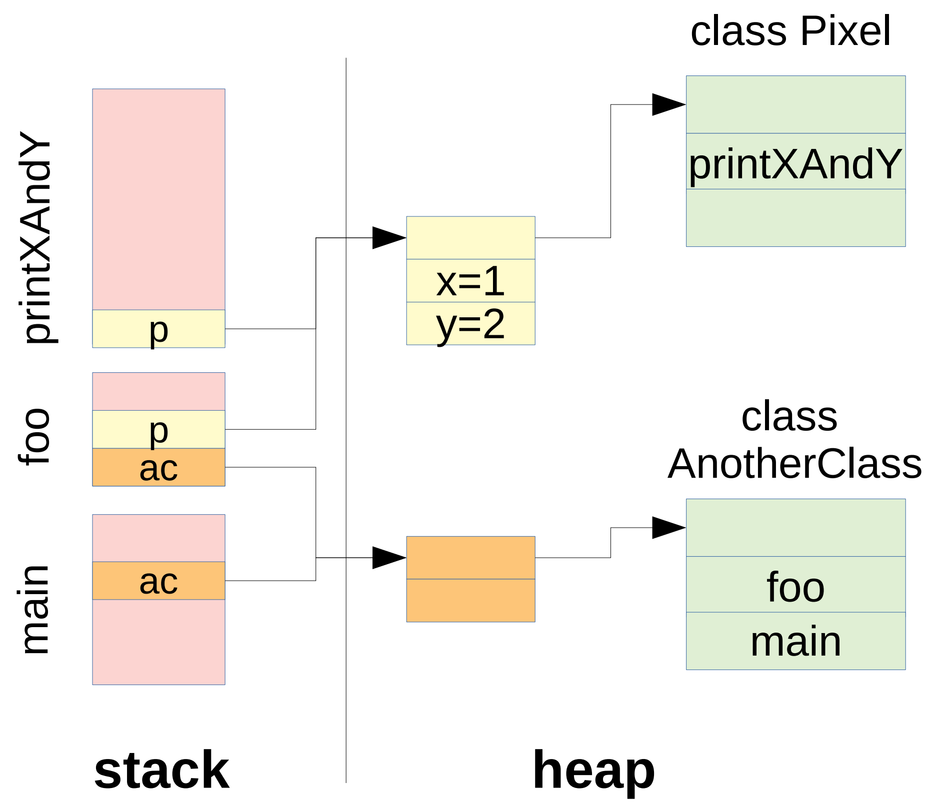
What a method call does

A method call copies arguments in parameter variables

The receiver object reference is copied into **this**

```
class Pixel {
  int x;
  int y;

  void printXAndY() {
    System.out.println(this.x
                 + " " + this.y);
  }
}
class AnotherClass {
    void foo() {
        Pixel p = ...
        p.printXAndY();
    }
    ... void main(...) {
        AnotherClass ac = ...
        ac.foo();
    }
}
```



Sometimes there's no **this**

`printXAndY()` is called on `p` (**`this`** value is those of `p`)
`foo()` is called on `ac` (**`this`** value is those of `ac`)

But on which reference `main()` is called?

- `main()`'s execution does NOT rely on any object, any instance
- it only relies on the **class** `AnotherClass` itself
- this method is said to be **`static`** and is "called on" the **class** rather than on an object (just like `Math.`*`sqrt`*`()`)
  - The use of **`this`** is forbiden in its code

```
class AnotherClass {
   public static void main(String[] args) { ... } // entry point
...
public final class Math {
  public static double sqrt(double a) { ... }
...
```

The `main` method

In Java, a class defining a method main with the (exact) following signature:

```
public class Pixel {
   public static void main(String[] args) {
      ...
   }
}
```

could be “executed”, i.e. called from command line

```
user@home$ java Pixel
```

The **`java`** command starts a Java Virtual Machine (JVM) and asks it to execute the method main of the class Pixel

But **this** could also be implicit

```
class Pixel {
   int x;
   int y;
   void printX() {
      System.out.println(x);
      // equivalent to
      // System.out.println(this.x);
   }
   void printY() {
      System.out.println(this.y);
   }
   void printXAndThenY() {
      printX();
      printY();
      // equivalent to
      // this.printX();
      // this.printY();
   }
}
```

Either for field

or for method call

Method call vs function call

In C, we wrote functions:

```
distance(p1,p2); // ask for distance between p1 and p2
```

Where should we define the function? who’s responsible?

In Java, we write **methods**:

```
p1.distance(p2); // ask p1 for its distance to p2
```

The method must be in the class Pixel (of `p1`), which is responsible for calculating the distance to any another point

Since a method is bound to a class, it must be called

either on an object (of the class in which it is defined)

```
p1.printXAndY(); // display coordinates of p1
```

or on the class itself

```
Math.sqrt(x*x + y*y)
```

Example

Class `Utils` is not intended to create instances; it is rather a “container” for static methods.

```
public class Utils {

  static int sum(int[] array) {
    var sum = 0;
    for(var value: array) {
      sum += value;
    }
    return sum;
  }

  public static void main(String[] args) {
    var array = new int[] { 1, 2, 3, 4, 5 };
    System.out.println(Utils.sum(array)); // 15

    // Utils. could be implicit (not recommended)
    // System.out.println(sum(array));
  }
}
```

No instance is required to call the method

Note: since Java 10, local variables could be declared with “var” keyword instead of a true type: compiler infers (guesses) the correct type

Naming conventions

Class names start with an UpperCase

method, field and variable names start with a lowercase

Names are build following the CamelCase convention

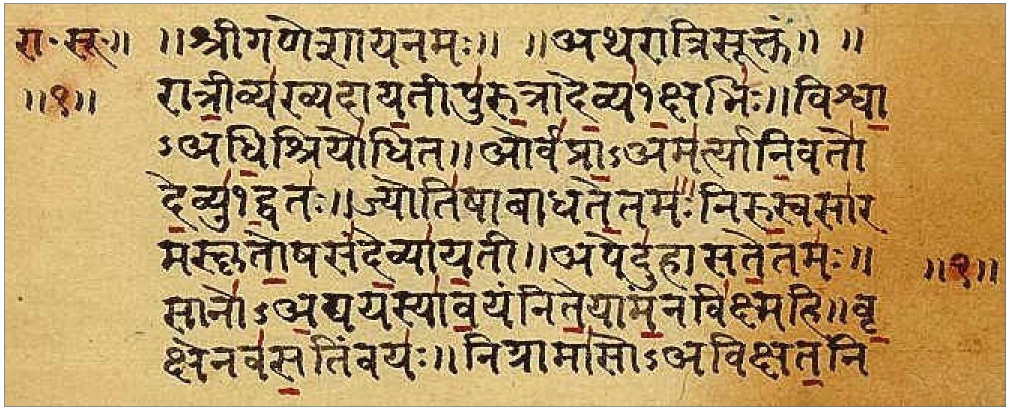
```
ThisIsAClass, thisIsAField, orALocalVariable,
orElseAParameter, andThisIsAMethod()
```

Underscore is only used for constant names

```
THIS_IS_A_CONSTANT
```

All names are in english!

Neither french, polish, tamil nor sanskrit



What is a class?

A **compilation unit**

compiling a file that contains a class of name `Toto` will generate a new file `Toto.class` with its **bytecode**

A **type** definition

used to declare variables or fields like `Toto t;`
also defines which methods are available for this type

A **mould / pattern** for the creation of instances/objects of this class

Based on the declaration of fields to be stored in its objects

It also defines the behavior (code) of methods

Class structure

A class is defined by its complete name (FQDN)

- Each class belongs to a **package** (no package = “default” package)
- `java.lang.String, java.util.List, fr.uge.imac.Example`

A class contains three kinds of **members**

- **Fields**, or attributes
- **Methods** and constructors
- **Inner classes**

Some members are **static**

- They are related to the class itself, and not related to an object
- Non static members cannot exist / have sense without an object

```java
package fr.upem.lecture;
public class Pixel {
    public final static int ORIGIN = 0;
    private int x;
    private int y;
    public Pixel(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void reset() {
        x = ORIGIN;
        y = ORIGIN;
    }
    public void printOnScreen() {
        System.out.println("("+x+","+y+")");
    }
    public static boolean same(Pixel one, Pixel two) {
        return (one.x==two.x) && (one.y==two.y);
    }
    public static void main(String[] args) {
        var p1 = new Pixel(1,3);
        var p2 = new Pixel(0,0);
        p1.printOnScreen(); // (1,3)
        System.out.println(Pixel.same(p1,p2)); // false
        p1.reset();
        System.out.println(Pixel.same(p1,p2)); // true
    }
}
```

Belonging package

Constant

Fields

Constructor

(instance) methods

Parameters

(class/static) methods

Local variables

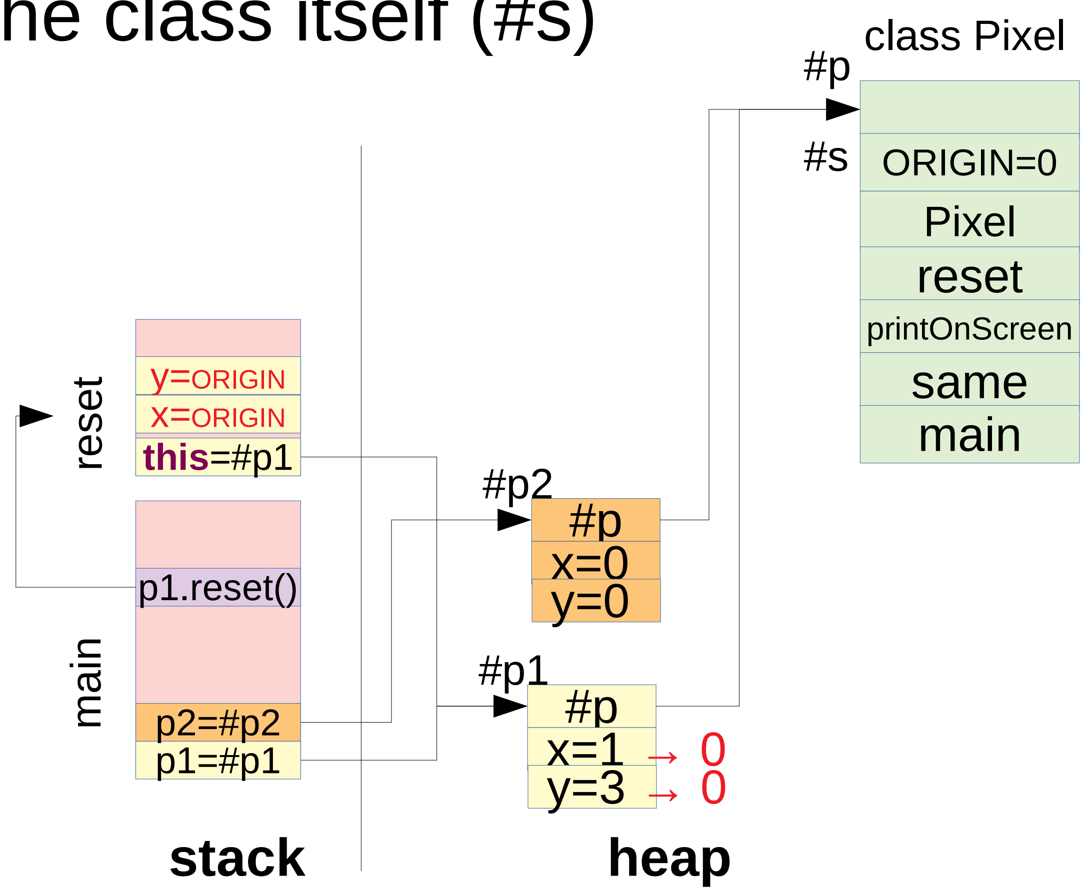
Arguments

When `p1.reset()` is invoked

its code is executed on the top of the stack, with **`this`** being the value of `p1` (#p1)

`ORIGIN` (static) is stored in the class itself (#s)

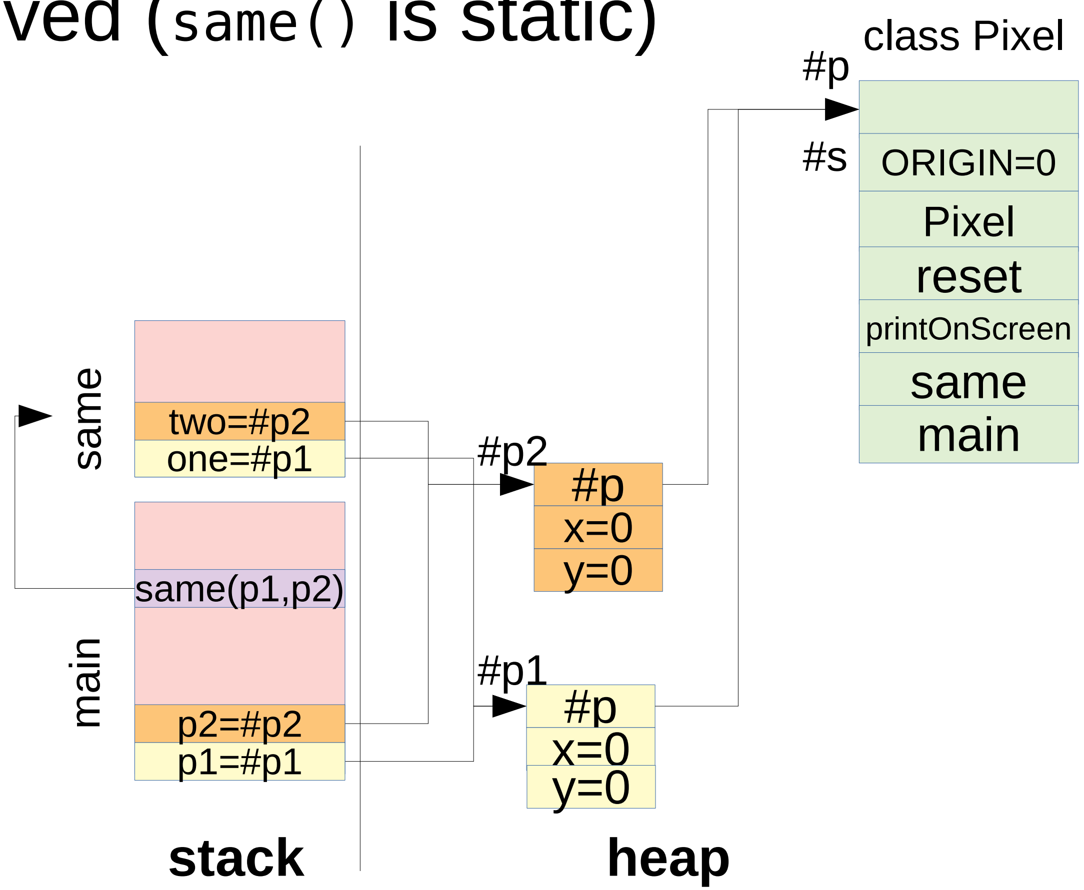
```
public void reset() {
    x = ORIGIN;
    y = ORIGIN;
}
// means
public void reset() {
    this.x = Pixel.ORIGIN;
    this.y = Pixel.ORIGIN;
}
// that is executed as
public void reset() {
    #p1.x = #s;
    #p1.y = #s;
}
```



When `Pixel.same(p1,p2)` is invoked

its code is executed on the top of the stack
values of p1 and p2 are copied in one and two
No **this** parameter is involved (`same()` is static)

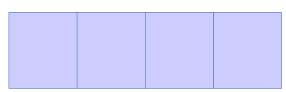
```
public static boolean
    same(Pixel one, Pixel two) {
    return (one.x==two.x)
        && (one.y==two.y);
}
// is executed as
public static boolean
    same(Pixel one, Pixel two) {
    return (#p1.x==#p2.x)
        && (#p1.y==#p2.y);
}
```

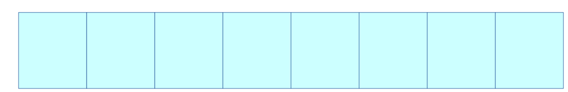


Two main sorts of types in Java

Fields or local variables in Java are of one of two sorts

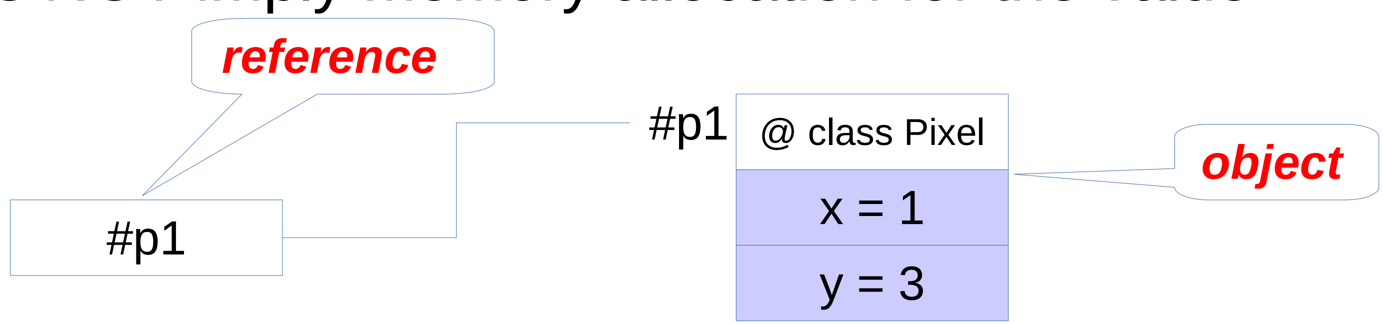
- **Primitive type**: **variable stores the value**
  - declaration implies memory allocation to store its value (depending on the type)

int myIntValue; 

long myLongValue; 

- **Reference type**, or “object” type: **variable stores reference to the value** (could be **null**)
  - declaration does NOT imply memory allocation for the value (like a pointer)

```
Pixel p1 = null;
p1=new Pixel(1,3);
```



Primitive types in Java

Signed integer types (in two's complement representation)
https://en.wikipedia.org/wiki/Two%27s_complement

- **byte**: 8 bits [-128 .. 127]
- **short**: 16 bits [-32768 .. 32767]
- **int**: 32 bits [-2147483648 .. 2147483647]
- **long**: 64 bits [-9223372036854775808 .. 9223372036854775807]

Default type for integer literals (1_000 is of type int, but 1_000L is of type long)

Unsigned character type (UTF-16 code units)

- **char**: 16 bits ['\u0000' .. '\uffff']

Flotting point types (IEEE 754 representation)
https://en.wikipedia.org/wiki/IEEE_754

- **float**: 32 bits
- **double**: 64 bits

Default type for floating point literals (3.14 is of type double, but 3.14F is of type float)

Boolean type

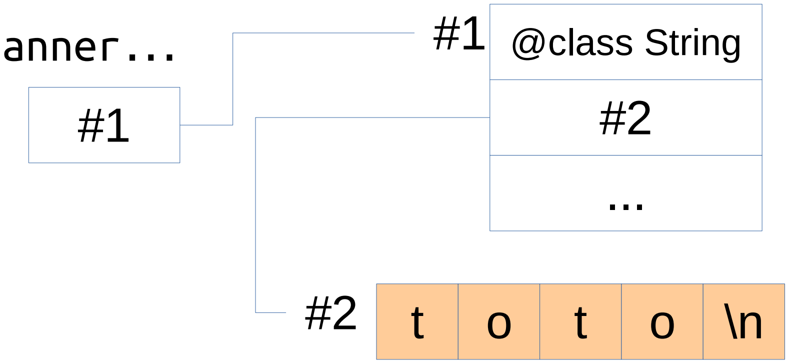
- **boolean**: (true / false)

All other types are “reference types”

API defined types (Application Programming Interface)

`java.lang.Object, java.lang.String, java.util.Scanner...`

```java
String s = "toto";
```

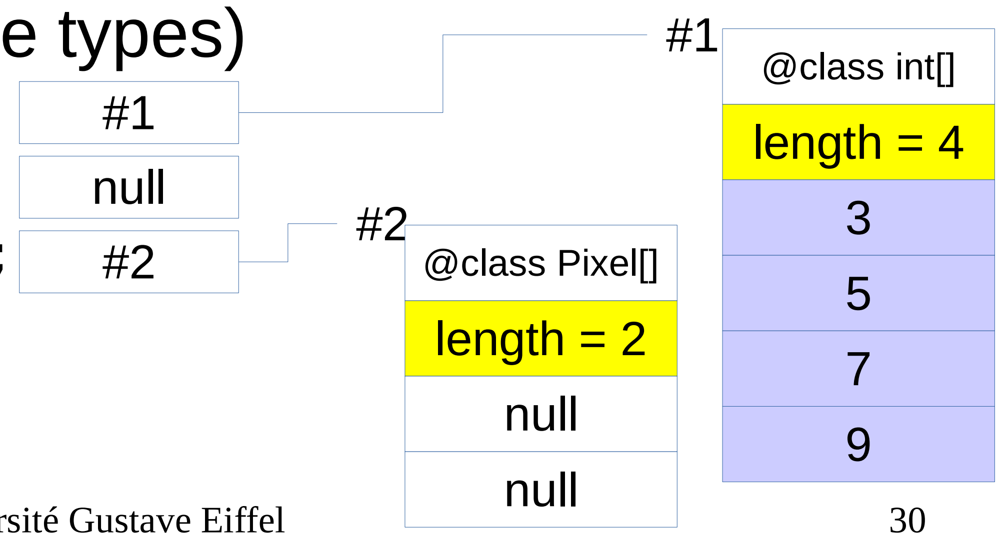


String are different from C
and they are constant (immutable)!

“Hidden” types of the language

Arrays (of primitive or reference types)

```java
int[] tab = {3,5,7,9};
String[] strings;
Pixel[] array = new Pixel[2];
```



User defined types

```java
Pixel p = new Pixel(0,0);
```

The `null` value

When declaring a reference type variable, its default value is **`null`**, a special value whose access is prohibited

Compiler try to avoid it

```
Pixel p;
p.printOnScreen();
```

Compiler signals:
"The local variable p may not have been initialized

JVM throw an exception

```
Pixel p = null;
p.printOnScreen();
```

Compiler is ok
but JVM throws
a NullPointerException

Default value?

Local variables (or parameters) live in the **stack**
Their lifetime is the method call

- Compiler requires they are initialized before to be used

Fields live in the **heap**
Their lifetime is those of their object

- When an object is created/allocated, its field are initialized by default to
  - `0` (or `0.0`) for primitive numeric types
  - **`false`** for booleans
  - **`null`** for any other reference type

> This is the job of the "constructor" (see later)

Memory allocation

To assign a value other than **null** to a variable, we need a “valid” reference (to a reserved memory part of the heap)

Such a valid reference is provided by the **new** operator

- **new** needs to know the size to allocate (like malloc)
- it’s given by the **type name** that follows the operator
  - Which is also the “constructor” name

```
var p = new Pixel();
```

Pixel class knows that 2 int fields are required to store each of its instances

- For arrays, the number of elements is required

```
var array = new int[10];
var array = new Pixel[10];
```

Either 10 times the size of a primitive (here int), or 10 times the size of a reference (for any other reference type)

Memory (de-)allocation

The new operator delegates memory management to the JVM

Similarly, memory liberation is managed by the JVM (its Garbage Collector or GC)

- Memory of objects that are no more used could be recycled and then available for new objects.
- A variable stops to reference (use) an object when
  - we leave the block where it was defined on the stack: it dies, disappears
  - It is assigned to an other value (either on the stack or the heap)

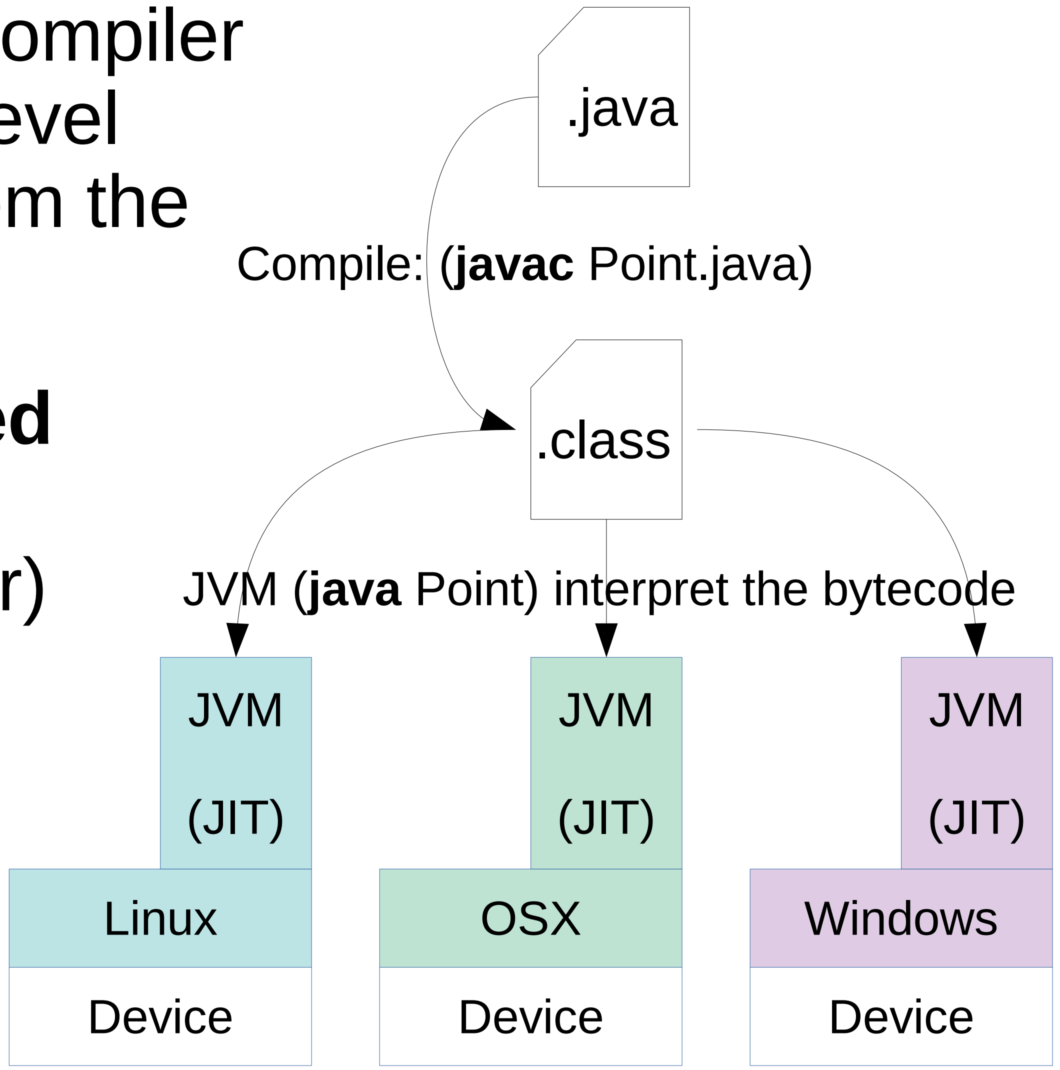
We could help the GC by explicitly assigning null to variable referencing objects we do not use anymore

Java execution model

From java source file, the compiler produces **bytecode** (high level assembly), independent from the host execution system

This bytecode is **interpreted** by any JVM for its host OS
A JIT (Just In Time compiler) optimizes execution

JVM implementations are provided for each host execution system



Encapsulation

“The only way to change the state of an object is to use its methods”

=> Limits the access/modification of a field (object state) to a small amount of code

Indeed, only methods of its class (in the same file!)

=> Helps programmer to guarantee invariant

For instance, field x is always positive

=> allows side effects to be controlled

A founding principle of OOP

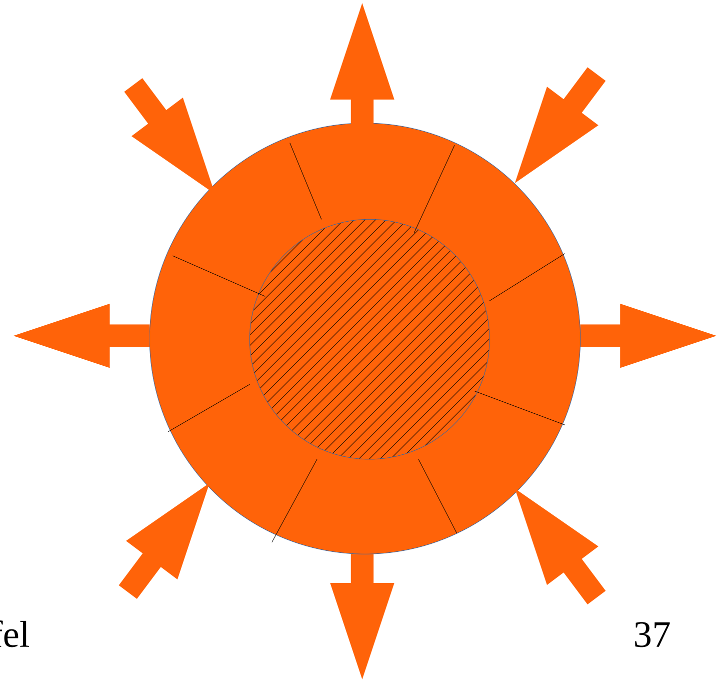
Helps design: one responsibility / one object

Helps debug: modifying code is local

Helps maintenance (correction/evolution)

Accessibility of the object's inside is restricted to the methods of its class

The interface of the object (interaction with outside) is the set of its **public** methods



How to code encapsulation?

Declare all the fields with the keyword `private`

Prohibits their access outside the class

Declare a method with the keyword `public` iff it stands for a required functionality (for outside)

`private` otherwise
avoid to give access to something not required!

```
public class Pixel {
   private int x;
   private int y;
   public double distance() {
      return Math.sqrt(x*x + y*y);
   }
   private double theta() {
      return Math.atan2(y, x);
   }
   ...
}
```

If some **internal** code requires trigonometry stuff

Three level of accessibility in Java

In the **class**: members (fields, methods, inner class) could be

- `private` : accessible only in this class
- *default* (no modifier): accessible from this package's classes
- `protected` : *default* + inherited classes in other packages
- `public` : accessible from anywhere the class is accessible

In the **package**: classes could be

- *default* : accessible only from this package's classes
- `public` : accessible from anywhere the package is accessible

A **module** explicitly **exports** accessible packages
http://tutorials.jenkov.com/java/modules.html

Constructor and object creation

An object (instance) is created in three steps:

```
var p1 = new Pixel(1,3);
```

The operator **`new`** asks for the JVM for a memory zone (the size is known thanks to the following class name)

- Each field is assigned to the default type value (`0`, **`false`**, **`null`**)
- A initialization block could be executed

The class name is also the name of the **constructor** of the class (kind of special method used to initialize the object, based on potential parameters)

`new` returns the reference to the memory zone

Allocation and initialization are critical operations for objects

If initialization relies on a “init” method to be called after allocation (like after malloc in C), it could be forgotten

```
public class Calc  {
  private int divisor; // required invariant : « divisor!=0 »
  public void init(int divisor) { // simple method initialization
    if (divisor == 0) {
      throw new IllegalArgumentException("divisor cannot be null");
    }
    this.divisor = divisor;
  }
  public double divide(int value) {
    return value / divisor;
  }
}
```

“default” constructor

```
public static void main(String[] args) {
  Calc c = new Calc();
  c.init(3);
  var res = c.divide(15);
}
```

What if this call is forgotten?

Constructors allow initialization to be guaranteed

Constructor is a compulsory entry point

Indeed, an object cannot be created without executing a constructor of its class (contrarily to 'init' after malloc in C)

```
public class Calc  {
  private int divisor; // required invariant : « divisor!=0 »
  public Calc(int divisor) { // constructor mandatory initialization
    if (divisor == 0) {
      throw new IllegalArgumentException("divisor cannot be null");
    }
    this.divisor = divisor;
  }
  public double divide(int value) {
    return value / divisor;
  }
}
```

```
public static void main(String[] args) {
  Calc c = new Calc(3);
  var res = c.divide(15);
}
```

Initialization cannot be forgotten

Constructor

Kind of “special method”:

- Same name as the class, **no return type**
- Cannot be called without **new** operator

If none explicitly defined, compiler adds one “default”

- without parameter

If at least one is explicitly defined, compiler does not add anything

```
public class Box {
  private int field;
  public static void main(String[] a){
    Box b = new Box();  // OK
  }
}
```

```
public class Box {
  private int field;
  public Box(int field) {
    this.field = field;
  }
  public static void main(String[] a){
    Box b = new Box();   // undefined
    Box b = new Box(2);  // OK
  }
}
```

Constructor overloading (“*surcharge*”)

Several constructors could be defined

- Overloaded to offer additional initialization services

Generally, one is the “most general”

- the others should refer to avoid code duplication

Use `this()` to call a constructor from an other one

- Do not use `new` (no need to re-allocate!)

```
public class Pixel {
  private int x;
  private int y;
  public Pixel(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Pixel() { // origin
    this(0,0);
  }
  public Pixel(int v) { // diagonal
   this(v, v);
  }
}
```

final fields

To guarantee invariants after object creation, we could ensure the fields will never change

If a field is declared **final**, then compiler will check that it is assigned **once and only once**, whatever the constructor used

private and **final** are recommended field modifiers to prevent side effects

```
public class Pixel {
  private final int x;
  private int y;
  public Pixel(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Pixel() {
    // error:  final field x may not
    // have been initialized
  }
  public Pixel(int v) {
   this(v, v);
  }
  public static void main(String[] a){
    Pixel p = new Pixel(1);
    p.x = 0; // error: final field x
             // cannot been assigned
  }
}
```

`private` constructor

Some classes are not intended to create objects

- Defining its constructor(s) as **`private`** prevent any object creation outside the class

```java
public class Utils {
  private Utils() { }
  public static int sum(int[] array) { ... }
}
```

Also use when object creation must be performed by a **factory method**…

The code of a constructor must be simple (assignments)

- Difficult to debug something that is partially initialized

If complex initialization code is required, prepare it apart of the constructor itself

Factory example

To avoid complex computations in the unstable initialization phase of an object creation

```
public class Box {
  private int field;
  public Box(int param) {
    // oh no !!
    // a complex code that uses
    // param to compute field
    field = ...
  }
}
```

```
public static void main(String[] a) {
  var b = new Box(3);
}
```

Factory example

To avoid complex computations in the unstable initialization phase of an object creation

```java
public class Box {
  private int field;
  public Box(int param) {
    // oh no !!
    // a complex code that uses
    // param to compute field
    field = ...
  }
}
```

```java
public class Box {
  private int field;
  private Box(int field) {
     this.field = field;  // cool
  }
  // factory method
  public static Box createBox(int param) {
    // a complex code that uses param
    // to compute field (in static context)
    var field = ...
    return new Box(field);
  }
}
```

Offer a public static factory method that prepares computations and then calls the private constructor to create object

```java
public static void main(String[] a) {
  var b = Box.createBox(3);
}
```

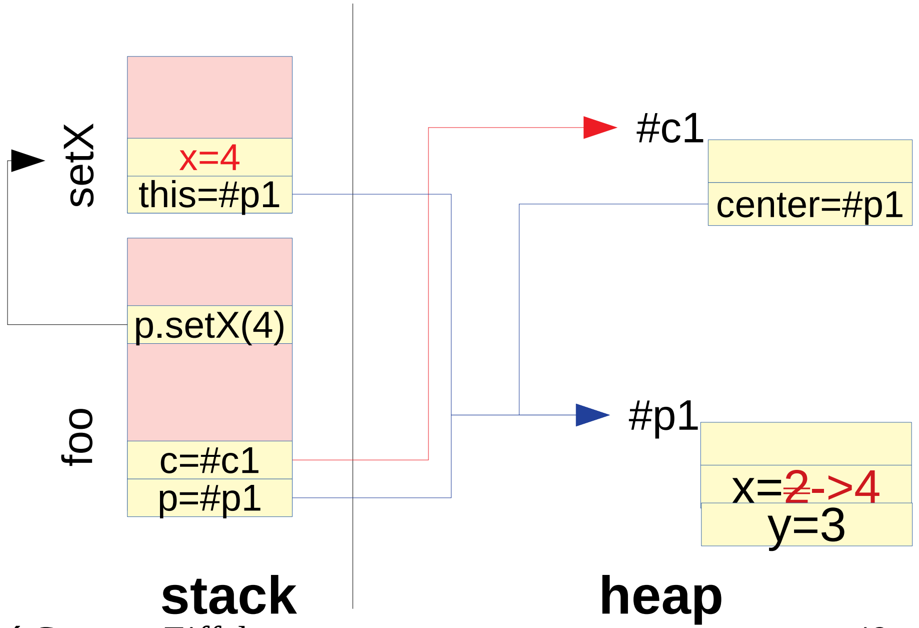
Encapsulation

[reminder] Encapsulation: the only way to change the state of an object is to use its methods

```java
class Point {
    private int x;
    private int y;
    public void setX(int x) {
        this.x = x;
    }
}
```

```java
class Circle {
    private final Point center;
    public Circle(Point center) {
        this.center = center;
    }
}
```

```java
class Usage {
    public void foo() {
        var p = new Point(2,3);
        var c = new Circle(p);
        p.setX(4); // Oups!
    }
}
```



Side effect is the problem

No side effect => no problem. So, avoid side effects.

Object’s state modification should imply new object creation

```
class Point {
    private int x;
    private int y;
    public void setX(int x) {
        this.x = x;
    }
}
```

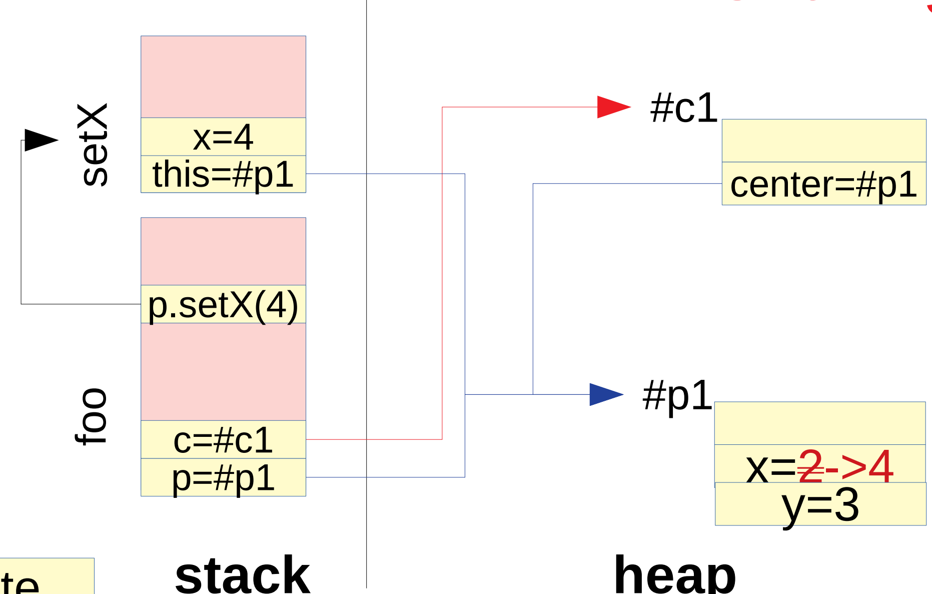
mutablility?

```
class Circle {
    private final Point center;
    public Circle(Point center) {
        this.center = center;
    }
}
```

sharing?

```
class Usage {
    public void foo() {
        var p = new Point(2,3);
        var c = new Circle(p);
        p.setX(4); // Oups!
    }
}
```

Change of the circle state
without invoking any
of its method



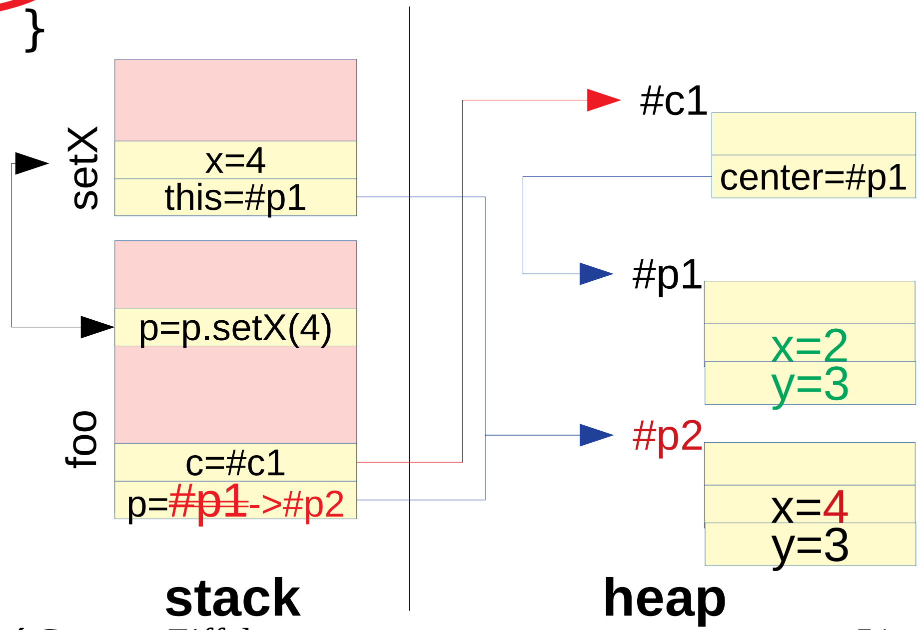
Solution 1: make `Point` immutable

The fields of objects cannot change their value

```
class Point { // immutable
    private final int x;
    private final int y;
    public Point setX(int x) {
        return new Point(x,this.y);
    }
}
```

```
class Circle {
    private final Point center;
    public Circle(Point center) {
        this.center = center;
    }
}
```

```
class Usage {
    public void foo() {
        var p = new Point(2,3);
        var c = new Circle(p);
        p = p.setX(4); // OK!
    }
}
```



Immutable class

A class is **immutable** if it does not allow its object state to change

- Unfortunately, in Java, it is not possible to enforce this by a keyword or through the compiler
- We have to **write** it down **explicitly** in the **documentation**

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

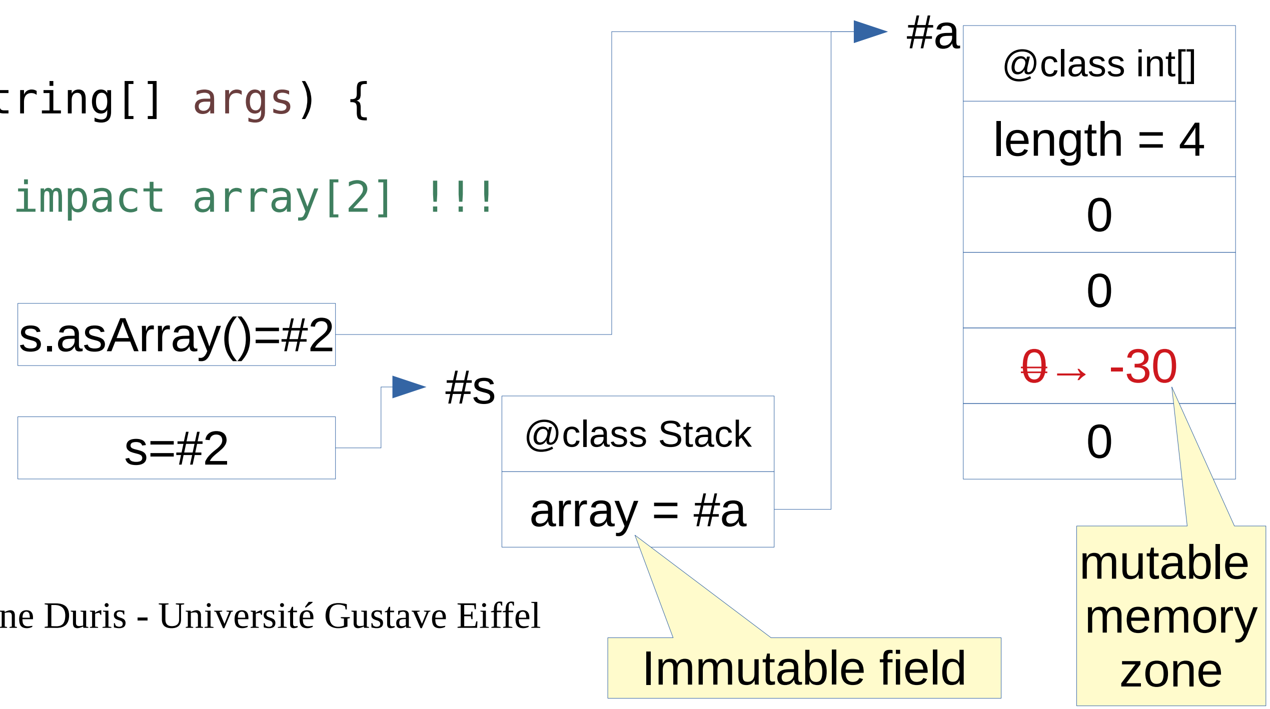
Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

- We could only enforce that (all) the fields could not be modified (either of primitive type, or reference type). Not sufficient!

Arrays (elements) are always mutable

Immutable reference type fields could point to mutable memory zones

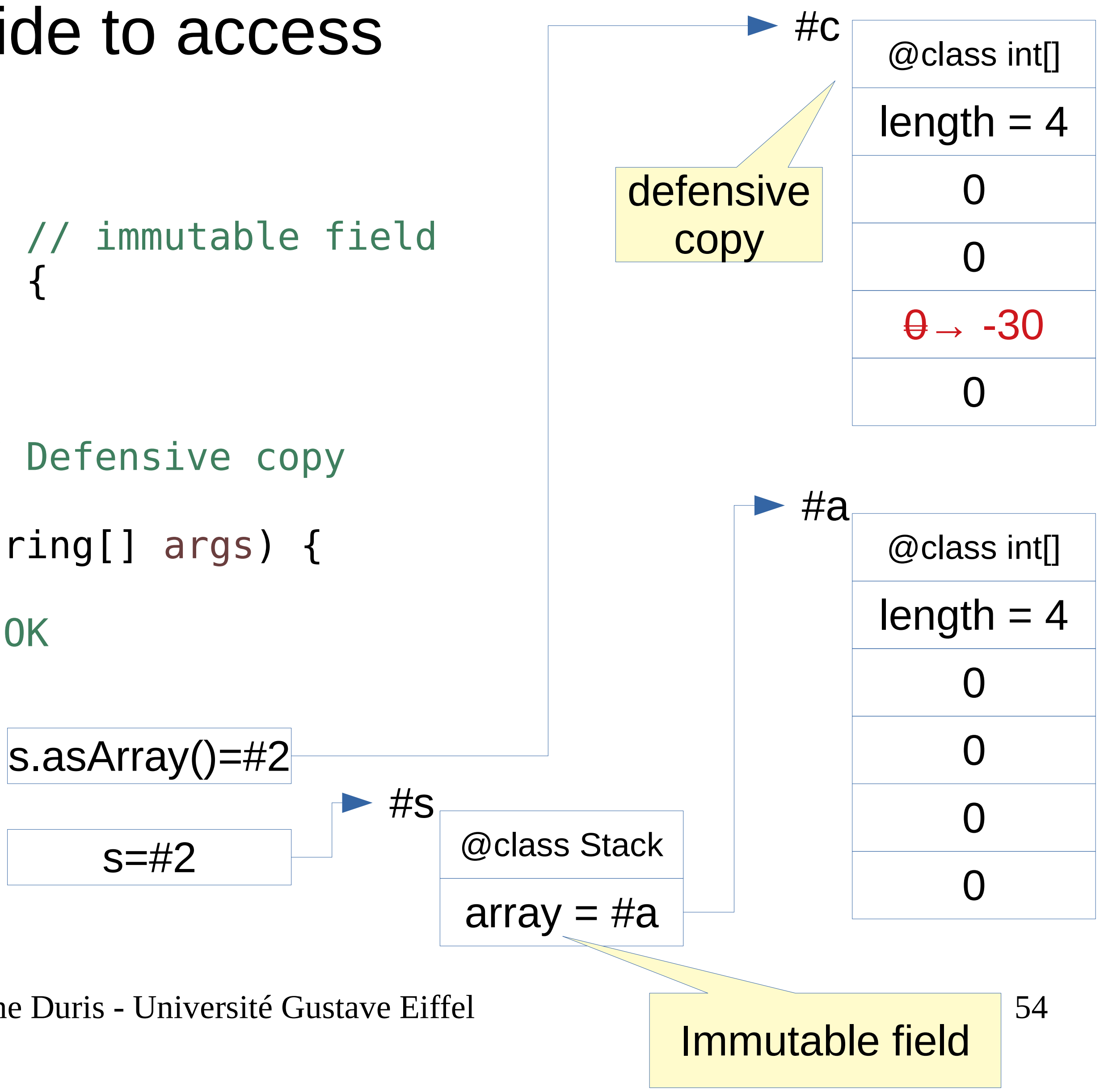
```java
public class Stack {
  private final int[] array; // immutable field
  public Stack(int capacity) {
    array=new int[capacity];
  }
  public int[] asArray() {
    return array;
  }
  public static void main(String[] args) {
    Stack s=new Stack(4);
    s.asArray()[2]=-30;  // impact array[2] !!!
  }
}
```



Arrays (elements) are always mutable

We must prevent outside to access (and modify) inside

```
public class Stack {
  private final int[] array; // immutable field
  public Stack(int capacity) {
    array=new int[capacity];
  }
  public int[] asArray() {
    return array.clone(); // Defensive copy
  }
  public static void main(String[] args) {
    Stack s=new Stack(4);
    s.asArray()[2]=-30;  // OK
  }
}
```



How make a class immutable?

1. declare all its fields as `final`

2. arguments of the constructor(s) must be

   either value of **primitive** type

   or reference to an object of **immutable class**

   or if a reference to a mutable class, then make a **defensive copy** of this object/reference

3. If a field of a mutable class must be published outside, then give a **defensive copy** of this object/reference

Solution 2: with `Point` mutable

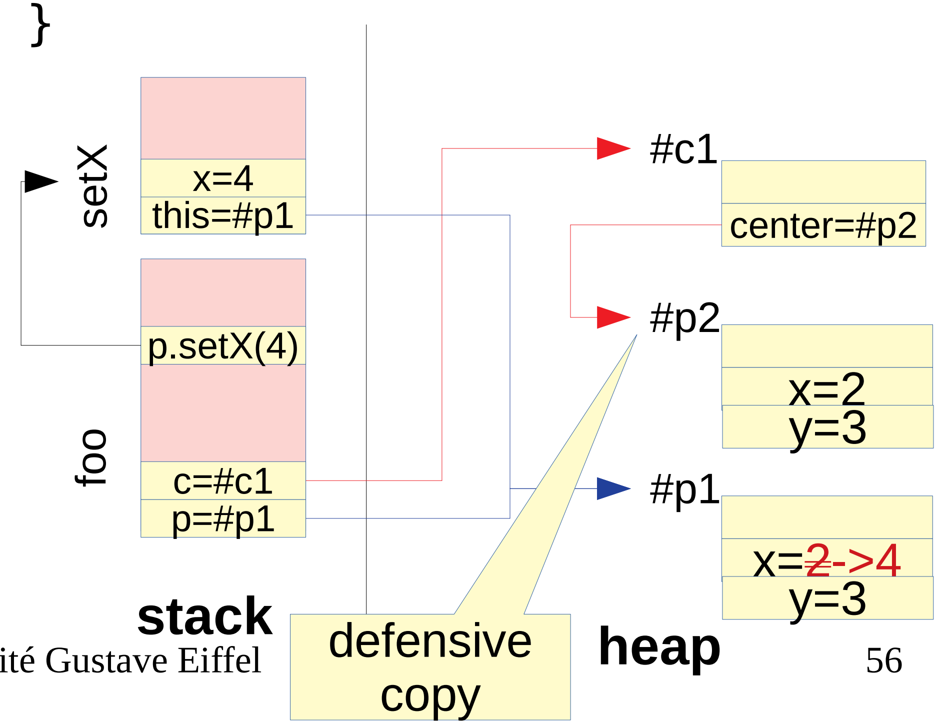
If class `Point` is mutable, class `Circle` must make **defensive copies** each time it exchanges with outside

```
class Point { // mutable
    private int x;
    private int y;
    public void setX(int x) {
        this.x = x;
    }
}

class Circle { // immutable
    private final Point center;
    public Circle(Point center) {
        this.center=center.clone();
    }

    // publish only defensive copy
    public Point getCenter() {
        return center.clone();
    }
}
```

```
class Usage {
    public void foo() {
        var p = new Point(2,3);
        var c = new Circle(p);
        p.setX(4); // OK
    }
}
```



mutable or not?

Usually

- Small objects could be immutable
  - Garbage Collector easily recycle them
- Bigger objects (arrays, lists, hash tables…) are mutable
  - For efficiency reasons

And if a field f of a class C is mutable, use defensive copy on f to make the class C immutable

Note: clone() requires some explanations… see later