Object, sub-typing & polymorphism

Etienne Duris

Université Gustave Eiffel - ESIPE

Plan

java.lang.Object

Sub-typing

Polymorphism

Overriding equals() and hashCode()

Records

Type, reference and objects

Two sorts of types in Java

Primitive type: **variable stores the “value”**

boolean, byte, char, short, int, long, float, double

Reference type, or “object” type: **variable stores a reference to the value**

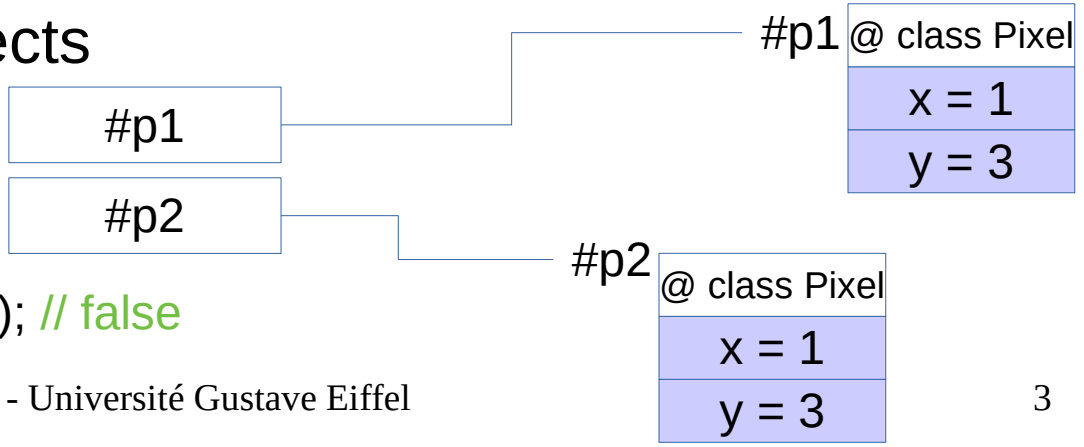
Object, String, arrays...

Operator **==** (primitive equality) concerns “values” or “references”

But not the referenced objects

```
Pixel p1=new Pixel(1,3);
Pixel p2=new Pixel(1,3);

System.out.printn(p1==p2); // false
```



java.lang.Object

This class is the “mother” of all others in Java

Any class “extends” java.lang.Object, directly or not

```
public class Person {
    private final String name;
    private final int birth;
    public Person(String name, int birth) {
        this.name = name;
        this.birth = birth;
    }
}
```

Implicitly extends Object
This means that all members (field, methods) of class Object are present (inherited) in class Person

A `Person` is an `Object` with something more (`name` and `birth`)

toString, equals() & hashCode()

Each of these methods has a default implementation in class Object, that is inherited in any class

```
Person mark = new Person("Mark", 1950);
mark.toString();
// Person@4b1210ee
// default: class+"@"+hashCode()
mark.equals("hello");
// false
// default: primitive equality (==)
mark.equals(mark);
// true
// default: primitive equality (==)
mark.hashCode();
// 1259475182, i.e. 0x4b1210ee
```

Methods in Object

`java.lang.Object` defines some universal methods

available on any object of an class

`toString()`

Returns a string representation of an object

`equals()`

Indicates whether some other object is "equal to" this one (if their states are equals): true or false

`hashCode()`

Returns a hash code value for the object (an int)

Why does it works?

```
Person mark = new Person("Mark", 1950);
System.out.println(mark);
// Person@4b1210ee
```

How it is possible to invoke a method of the JDK (`println`) with an object (`mark`) of a class (`Person`) that was unknown until compilation a few minutes ago?

In `PrintStream` which is the type of `System.out`, the method `println` expects an argument of type `Object`

```
public void print(Object obj)
```

It works because `Person` is a **sub-type** of `Object`
(thanks to inheritance)

Plan

java.lang.Object

Sub-typing

Polymorphism

Overriding equals() and hashCode()

Records

Sub-typing

`Person` is a sub-type of `Object`

- this relation comes from implicit/explicit **inheritance**
- interface **implementation** also provides sub-typing

Basic principle of sub-typing:

everything we know to do on a type,
we know to do it on a sub-type

- It usually relies on methods, that stands for behavior (what we know to do)

An inheritance example

Let us consider a class Pixel

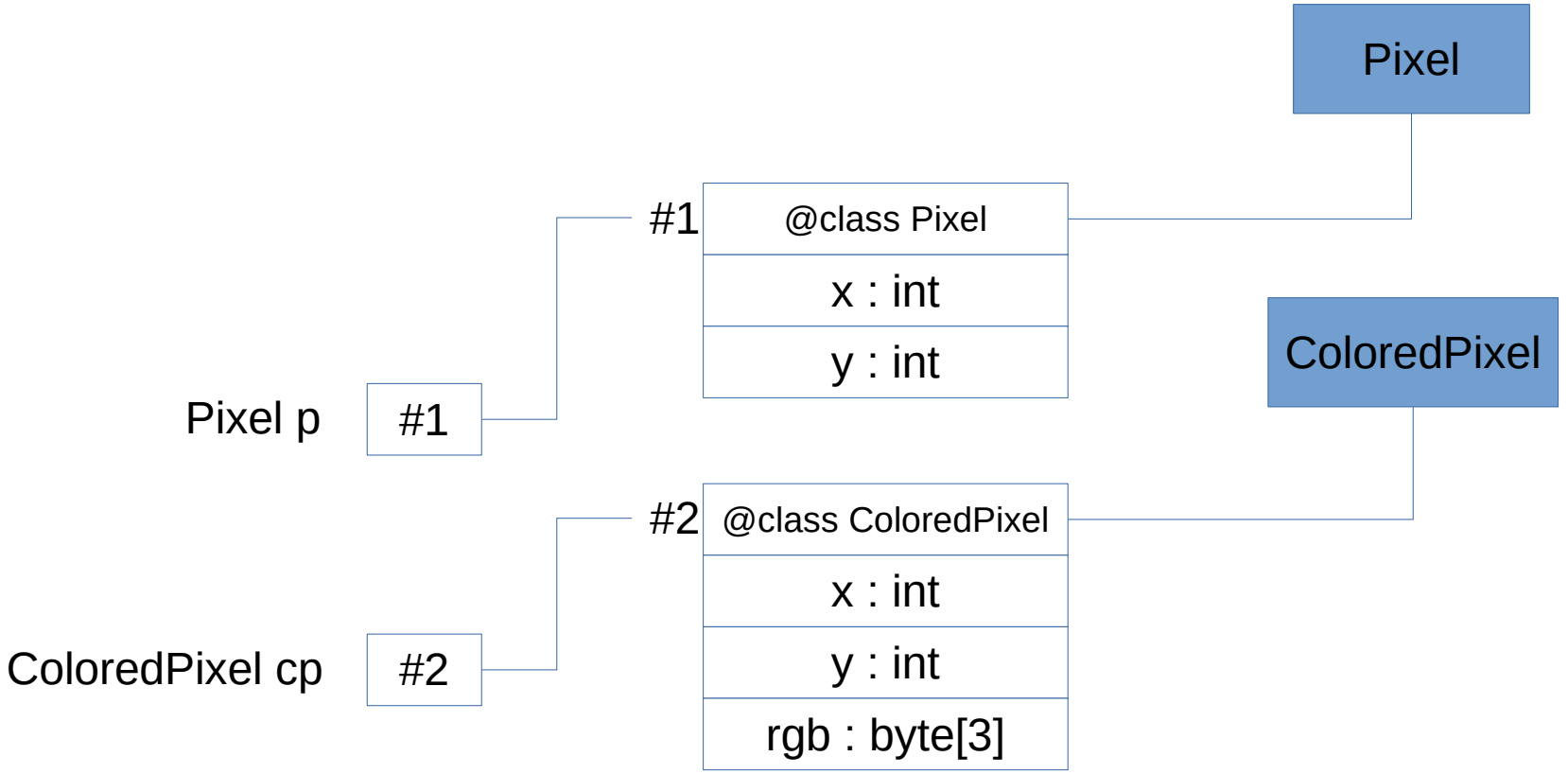
- Each instance of a Pixel has an x and a y of type int
- We know how to move a Pixel, with method
  void moveTo(int nx, int ny)

We now want to have a class ColoredPixel

- Each instance of a ColoredPixel has an x and a y of type int and also a color as an array of 3 bytes rgb
- We want to move a ColoredPixel with moveTo(int nx, int ny)
- We also want to get its color components with getRed(), getGreen() and getBlue()

From the memory point of view

An object of class ColoredPixel is just as an object of class Pixel, but with an additional field byte[] rgb



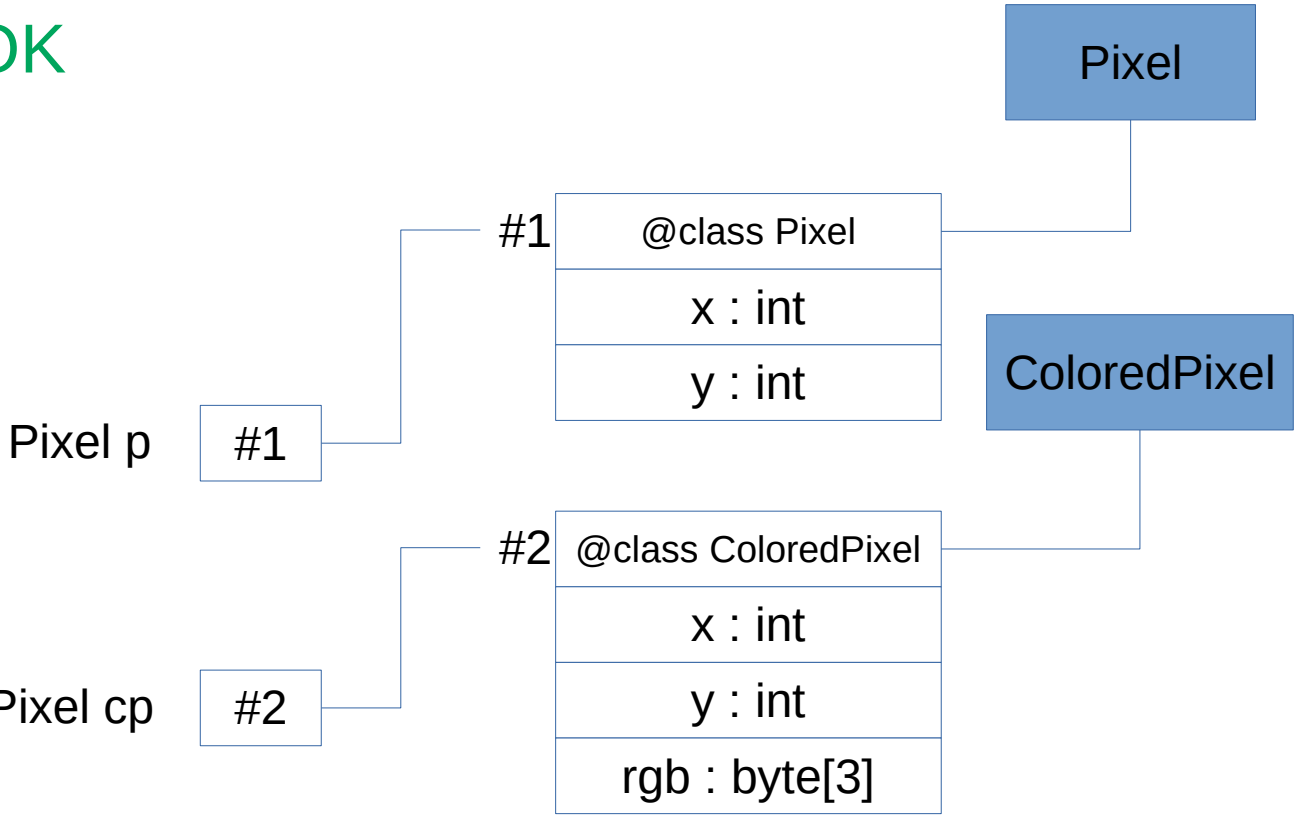
From the behavior point of view

Everything we know to do on a Pixel, we know to do it on a ColoredPixel

```
p.moveTo(2,2);   // OK
cp.moveTo(2,2); // OK
```

But not the contrary

```
cp.getBlue(); // OK
p.getBlue();   // KO
```



From the code point of view… without inheritance

```
public class Pixel {
    private int x;
    private int y;
    public void moveTo(int newX, int newY){
        this.x = newX;
        this.y = newY;
    }
}
```

```
public class ColoredPixel {
    private int x;
    private int y;
    private byte[] rgb;
    public void moveTo(int newX, int newY){
        this.x = newX;
        this.y = newY;
    }
    public byte getRed() { return rgb[0]; }
    public byte getGreen() { return rgb[1]; }
    public byte getBlue() { return rgb[2]; }
}
```

Duplication of field declarations

Duplication of method definitions

From the code point of view… with inheritance

```
public class Pixel {
    private int x;
    private int y;
    public void moveTo(int newX, int newY){
        this.x = newX;
        this.y = newY;
    }
}
```

Extends : « inherits from »

```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    public byte getRed() { return rgb[0]; }
    public byte getGreen() { return rgb[1]; }
    public byte getBlue() { return rgb[2]; }
}
```

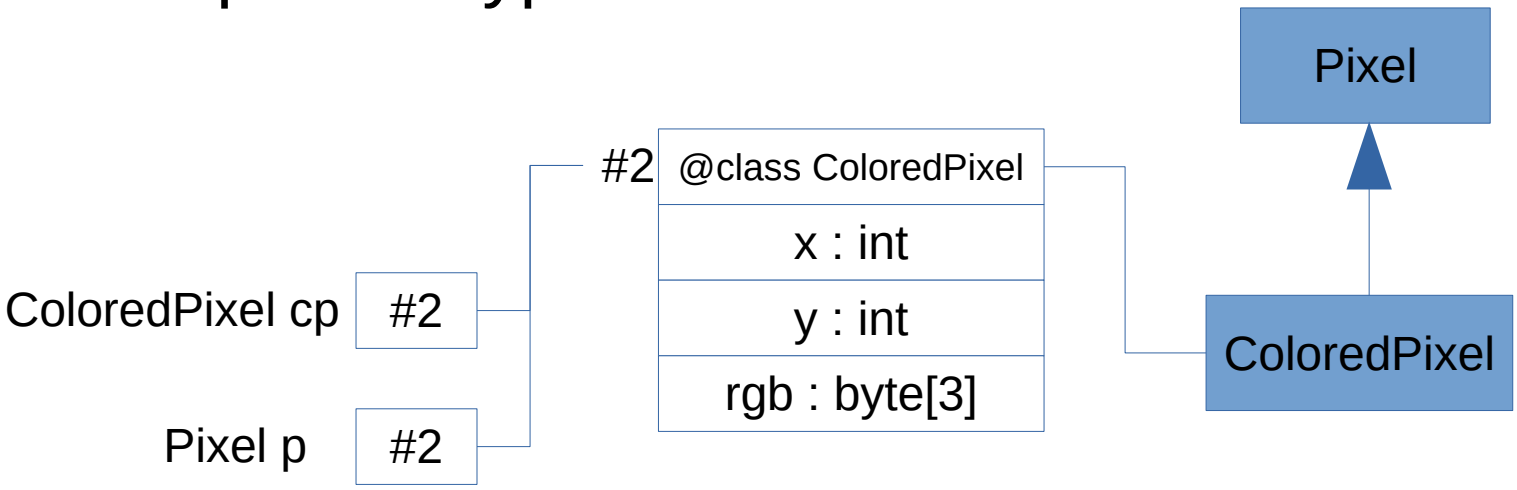
Fields (x and y) are inherited

Methods (moveTo) are inherited

Inheritance offers sub-typing

```
public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel();
    Pixel p = cp; // a ColoredPixel but could be handled as a Pixel
    p.moveTo(1,1);
}
```

The compiler accept to use a variable of a sub-type instead of the expected type



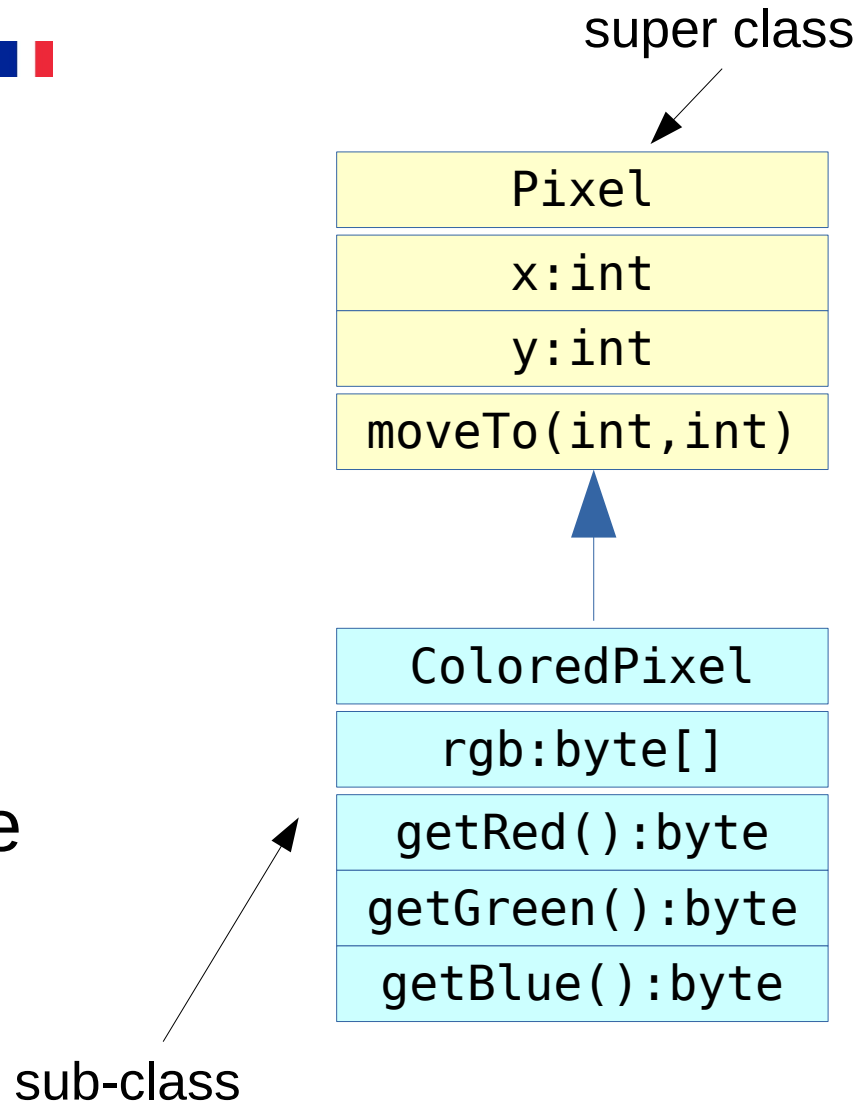
Vocabulary & notations

A sub-class **extends**
(or inherits from)
a super-class

hérite

The type defined by a sub-class
is a sub-type of the
type defined by the super-class

A reference to an instance
of a sub-class could be
stored and handled by a variable
defined with the super-type



Go back to our example: at compile time...

```
Person mark = new Person("Mark", 1950);
System.out.println(mark);
// Person@4b1210ee
```

Compiler agrees because `Person` is a sub-type of `Object` and because `Object x` can store `mark`

```
// in class PrintStream (of System.out)
public void println(Object x) {
  String s = String.valueOf(x);
  ...
}
```

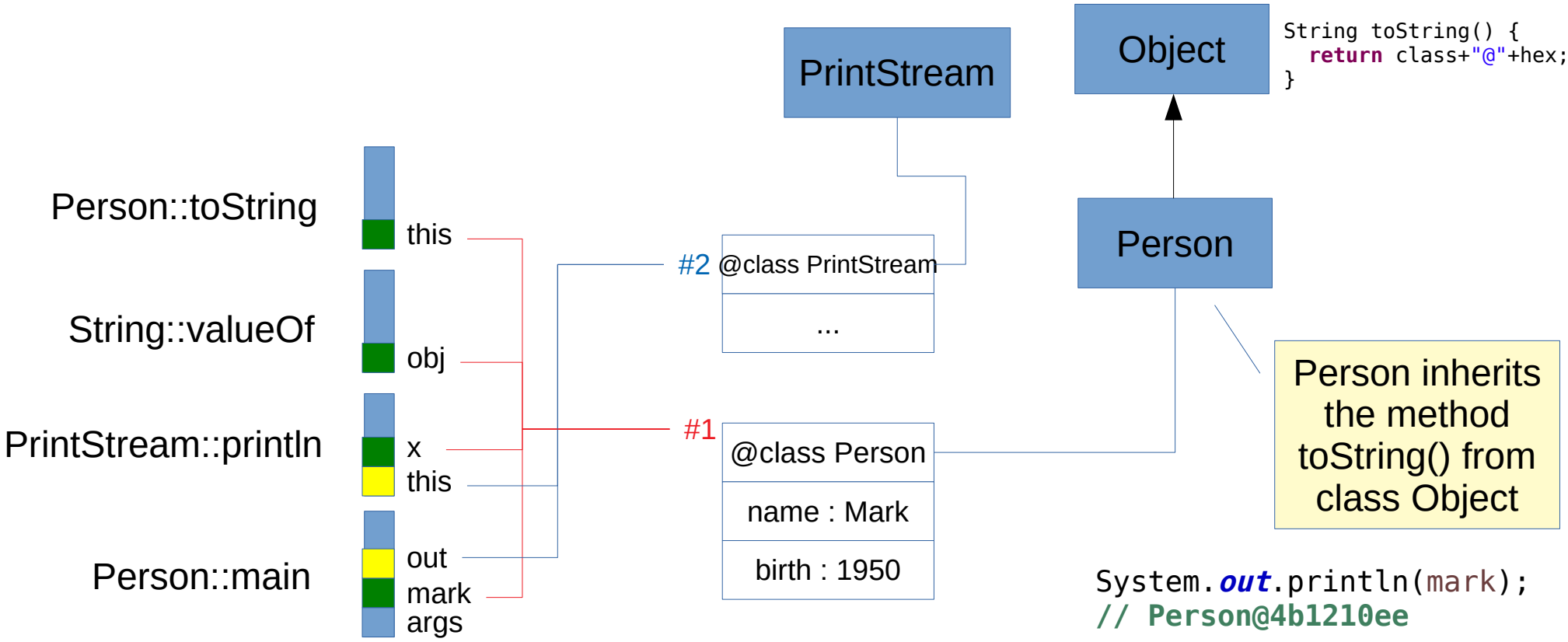
as `Object obj` in `valueOf()` stores `mark` to call `toString()`

```
// in class String
public static String valueOf(Object obj) {
  return (obj == null) ? "null" : obj.toString();
}
```

```
// in class Object
public String toString() {
  return getClass().getName()+"@"+Integer.toHexString(hashCode());
}
```

And at run time (execution)

The invocation of obj.toString() is made on a reference which, in fact, refers to an object of class Person



And what if...

with the same method call…

```
Person mark = new Person("Mark", 1950);
System.out.println(mark);
```

the class Person had its own toString() method:

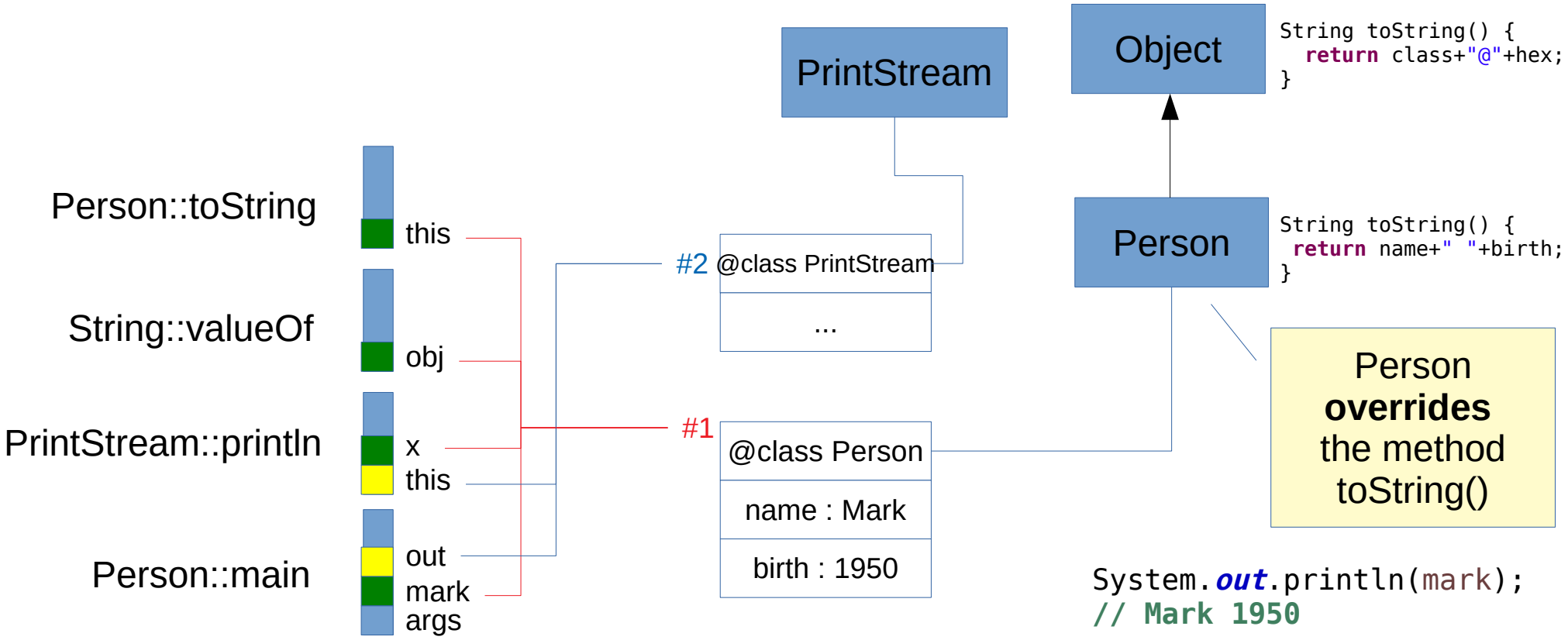
```
public class Person {
    private final String name;
    private final int birth;
    public Person(String name, int birth) {
        this.name = name;
        this.birth = birth;
    }

    public String toString() {
        return name + " " + birth;
    }

}
```

Polymorphism

Since the class Person has now its own toString() method definition, it is called at run time (instead of that of the class Object)



Plan

java.lang.Object

Sub-typing

Polymorphism

Overriding equals() and hashCode()

Records

Polymorphism

The substitution of one method call by another at run time depending on the class of the receiver object.

receiver.method(arg1, arg2);

```
Object[] array = new Object[] {
  "hello",
    new Object(),
    new Person("Mark", 1950)
};
for(Object value : array) {
  System.out.println(value.toString());
}
// prints
// hello
// java.lang.Object@4d7e1886
// Mark 1950
```

At compile time, compiler agrees since toString() exists for class Object (the **type** of the receiver)

At run time, the method toString() effectively called depends on the **class** of receiver object (String, Object, Person)

Sub-typing and polymorphism

Sub-typing allows us to reuse code (called "generic") that has been written by typing variable with a super-type (here Object) and calling the code, at run time, with a reference of a sub-type.

Polymorphism is the fact that when executing "generic" code with a reference on an object of a sub-class, a method invocation on a super-type variable effectively calls the method defined on the sub-type.

Method overriding (re-definition)

redéfinition 🇫🇷

A method overrides another if it could be substituted by polymorphism

For instance, Person::toString overrides Object::toString

Overriding a method is a way to **replace** an existing method definition (code) by a new definition (code) more suitable for a sub-type

Overriding and polymorphism

The JVM automatically performs polymorphism

- It cannot be avoided

Nevertheless, there is no polymorphism if there is no overriding:

- If the method is static (no object, no class at run time)
- If the method is private (not accessible)
- If method signature is not the same (see later…)
  for instance,
  public String toStrrring() { … } **overloads** *redéfinit*
  public String toString() (instead of **overrides**) *surcharge*
  => both methods are “applicable”

@Override : to avoid mistakes

@Override is an annotation that asks the compiler to verify that it does exist such a method to override in a super-type

```
// Compile time error:
// The method toStrrring() of type Person must
// override or implement a supertype method

@Override
public String toStrrring() {
  return name + " " + birth;
}
```

@Override is only used by the compiler (not by the JVM)

Even without @Override, polymorphism occurs

Plan

java.lang.Object

Sub-typing

Polymorphism

Overriding equals() and hashCode()

Records

equals() and ==

== tests reference identity

r1==r2 means both references point to the same memory address

boolean equals(Object) is defined in class Object to test structural identity of objects (if they have same the state)

Default implementation of equals in Object is **==**

Be careful: parameter type is **Object**, and must be **Object** in method redefinition

It is necessary to redefine equals(), because collections of java.util are using this method to search for objects

How to Override equals(Object o)

How access to the fields of the argument?

```
public class Car {
    private final String owner;
    private final int numberOfWheels;
    ...
    @Override
    public boolean equals(Object o) {
        ...
        How to know if o.numberOfWheels exists?
        ...
    }
    public static void main(String[] args) {
        Car c1 = new Car("Oui-oui", 4);
        Car c2 = new Car("Oui-oui", 4);
        c1.equals(c2);
        Person mark = new Person("Mark", 1950);
        c1.equals(mark);
    }
}
```

Compiler refuses

Compiler agrees

Cast, declared type, real type

transtypage

Reference cast is the fact to explicitly force the compiler to consider a reference as being of a given type (that is not necessarily the type of the referenced object)

The JVM verifies, at run time, that it is possible to handle the referenced object as such a type (it must be a super-type of the object class). If not, the JVM raises a **ClassCastException**

```
class A { }
class B extends A { }
class C extends B { }
```

```
B b = new B();
A a = b;
// B b2 = a;  // incompatible types
B b2 = (B) a; // OK
C c = (C) a;  // ClassCastException
```

```
Object o;
if(Math.random() > 0.5)
  o = "toto";
else
  o = new Object();
String s = (String) o;
// Compiler agrees but 50-50
// chance to raise a
// ClassCastException at run time
```

`instanceof` operator

To avoid cast exception, we could test **x instanceof T** where

x is a variable holding a reference or null

T is a type

the result is true if x is not null and could be assigned to a variable of type T without ClassCastException; else false.


```
class A { }
class B extends A { }
class C extends B { }
```

```
A ab = null;
System.out.println(ab instanceof A);  // false
ab = new B();
System.out.println(ab instanceof A);  // true
System.out.println(ab instanceof B);  // true
System.out.println(ab instanceof C);  // false
```

```
Object o;
if(Math.random()>0.5)
  o = "toto";
else
  o = new Object();
if (o instanceof String)
var s = (String) o; // OK...
```

instanceof vs getClass()

var.getClass() returns the run time type (Class) of the reference stored in the variable var

```
Object o = new Car("Oui-oui", 4);
o.getClass() == Car.class;
```

Be carreful:

if o is null, o.getClass() throws a NullPointerException whereas o instanceof T returns false

This is usually more strict that what we want for equals

Back to our Car example

```java
@Override
public boolean equals(Object o) {
    if (!(o instanceof Car)) {
        return false;
    }
    Car car = (Car)o;
    return numberOfWheels==car.numberOfWheels
        && owner.equals(car.owner);
}
```

MUST be Object

Test for class and sub-classes

Use fields rather than getters (that could be redefined)

To compare objects, delegate to their class (equals)

equals(), null and efficiency

a.equals(b) is not symetric if a or b is null

- If a is null, throw NullPointerException
- If b is null, return false

Method java.util.Objects.equals(a,b) allows us to test if two objects are equals or both null.

Be careful: equals could be expensive (in execution time)

- For instance, String::equals tests each character...

Overriding hashCode()

hashCode() returns an int that “resume” the object on which it is called

Since hashCode() is used by the collection API based on hashtables (java.util.HashMap and java.util.HashSet), it is important that the hash value spreads as possible the integer range.

hashCode() must be side effect free

- else, we could not retrieve it in the hashtable

Defining hashCode() on a mutable object is dangerous

- if the object is modified, its hashCode() changes...

hashCode and equals()

Two objects o1 and o2 such that
o1.equals(o2) must verify that
o1.hashCode() == o2.hashCode()

The contrary is not true:
if o3.hashCode() == o4.hashCode(),
then it is possible that o3 and o4 are not equal

It’s a collision

If you **override equals()**,
then you **must override hashCode()**
and conversely

Example

```
public class Car {
    private final String owner;
    private final int numberOfWheels;
    public Car(String owner, int numberOfWheels) {
        this.owner = Objects.requireNonNull(owner);
        this.numberOfWheels = numberOfWheels;
    }
    @Override
    public int hashCode() {
        return numberOfWheels ^ owner.hashCode();
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Car)) {
            return false;
        }
        Car car = (Car)o;
        return numberOfWheels==car.numberOfWheels
                && owner.equals(car.owner);
    }
```

Implementation of hashCode()

It’s often simple to use the static method
java.util.Objects.hash(Object… values)

```
@Override
public int hashCode() {
    return Objects.hash(numberOfWheels, owner);
}
```

Implementation can also use prime numbers to avoid collision

- have a look at implementation of String::hashCode

- and try in eclipse: Source > Generate hashCode() and equals()...

  - be careful to useless instructions

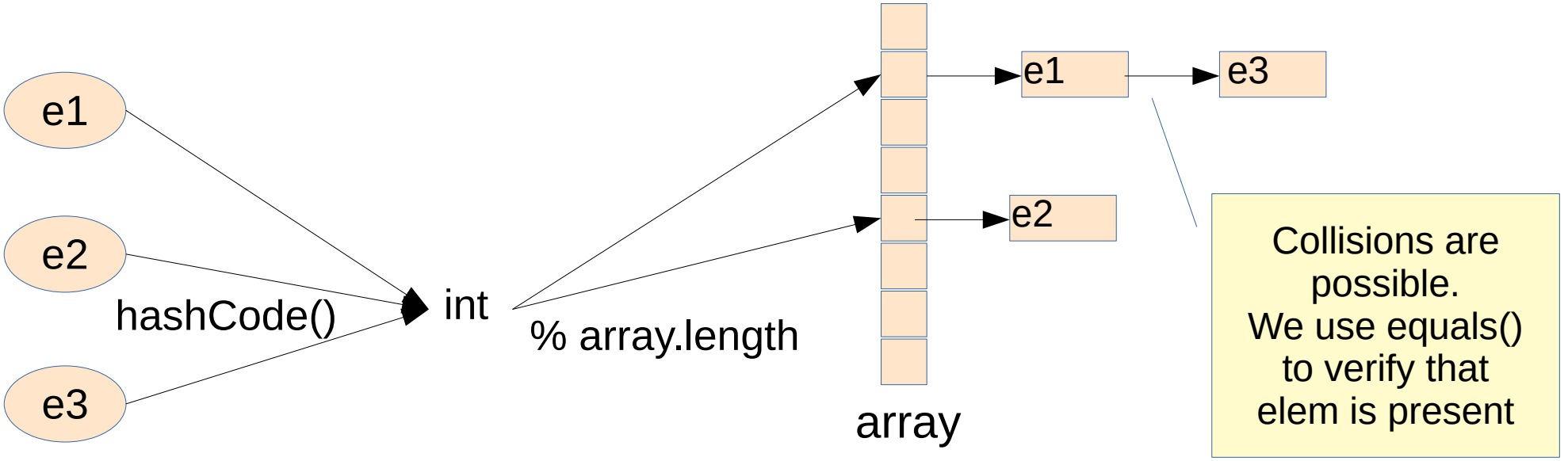
Recall about hash tables

ArrayList.remove(Object elem) or
ArrayList.contains(Object elem) are in O(n)

They have to scan all the elements of the list to find elem

HashSet.add(), remove() and contains() are in O(1)

We try to store elem in the right bucket
thanks to its hashCode



Switch on String

It is possible to use a switch statement with strings

```
switch(args[0]) {
case "-verbose":
   ...
   break;
case "-short":
   ...
   break;
case "-help":
   ...
   break;
default:
   ...
}
```

```
switch(args[0].hashCode()) {
    case 0x57825cb5: // "-verbose".hashCode()
       if (args[0].equals("-verbose")) {
          ...
       }
    case 0x534f986f: // "-short".hashCode()
       if (args[0].equals("-short")) {
          ...
       }
    case 0x2aaf0ee:  // "-help".hashCode()
          if (args[0].equals("-help")) {
          ...
```

Compiler computes hashCode() values for each case String (constants)

And then equals() is called to manage collisions

Plan

java.lang.Object

Sub-typing

Polymorphism

Overriding equals() and hashCode()

Records

record: a new way to create objects

"Preview" feature (since jdk14)

A "record" is a named tuple, non mutable

stands for data whose fields are known (not hidden)

Goal: minimize the code to write to build a class that holds for "*the fields, just the fields, and nothing but the fields.*" [B. Evans]

```
public record Car(String owner, int numberOfWheels) {

  public static void main(String[] args) {
    var c = new Car("Oui-oui", 4);
  }
}
```

`owner` and `numberOfWheels` are the "components" of the record `Car`
they are managed as private final fields

With a record, come for free...

A “canonical” constructor that we could override

```
public record Car(String owner, int numberOfWheels) {
    public Car(String owner, int numberOfWheels) { // canonical constructor
        this.owner = Objects.requireNonNull(owner);
        this.numberOfWheels= numberOfWheels;
    }
}
```

Default implementations for methods `equals()`, `hashcode()`, `toString()`

```
public static void main(String[] args) {
  var c1 = new Car("Oui-oui", 4);
  var c2 = new Car("Oui-oui", 4);
  System.out.println(c1.equals(c2)); // true
  System.out.println(c1.hashCode()); // -242146933
  System.out.println(c2.hashCode()); // -242146933
  System.out.println(c1); // Car[owner=Oui-oui, numberOfWheels=4]

  System.out.println(c1.numberOfWheels() + c2.numberOfWheels()); // 8
  System.out.println(c1.owner());    // Oui-oui
}
```

Getters

And some other features

Possibility to define a “compact” constructor (without arguments)

```
public record Car(String owner, int numberOfWheels) {
  public Car {  // compact constructor (no parameters)
    Objects.requireNonNull(owner); // no need to write “owner = “...
    if(numberOfWheels <=0) {
      throw new IllegalArgumentException("bad wheel number "
                                         + numberOfWheels );
    }
  }

  public Car(String owner) { // other overloaded contructors
    this(owner, 4);          // MUST first call this(...)
  }
}
```

Record cannot be extended

But record can implement interface… as we’ll see...