S7 Inf A3 — Java for Graphical and Mobile Programming · Centrale Lyon

Stéphane Derrode

CM3

Inheritance, Interfaces & Composition

- extends — inheritance & method overriding
- abstract classes — partial implementations
- interface — contracts & multiple implementation
- static fields & methods
- equals() / hashCode() from Object
- Composition vs inheritance

Module Overview

Part 1 — Java Fundamentals	Part 2 — GUI with Swing	Part 3 — Android
CM1 From Python to Java	CM4 Swing: Components & Layouts	CM6 Android Architecture
CM2 Collections, Exceptions & I/O	CM5 Graphics 2D & Events	CM7 Layouts, Navigation & Intents
CM3 Inheritance, Interfaces ◀ you are here		CM8 Persistence & Threads

Projects: ★ Spider Game (Part 2) ★ Calculator / Currency Converter (Part 3)

Exam 50% + BEs 50%

CM3 — Agenda

01 Inheritance — extends & super

02 Method overriding & polymorphism

03 Abstract classes

04 Interfaces

05 static fields & methods

06 equals() / hashCode() & composition

Inheritance — extends & super

```
// Base class — already know this
public class Voiture {
    protected boolean estDemarree; // visible to
subclasses
    protected double  vitesse;
    protected int     puissance;

    public Voiture(int p) {
        puissance   = p;
        estDemarree = false;
        vitesse     = 0.0;
    }
    public void demarre() {
        estDemarree = true;
    }
}

// Subclass — inherits all non-private members
public class VoitureElectrique extends Voiture {
    private boolean disjoncteur;

    public VoitureElectrique(int p) {
        super(p);            // MUST be first statement
        disjoncteur = false;
    }
    public void activerDisjoncteur() {
        disjoncteur = true;
    }
```

extends keyword

A class can extend exactly ONE other class. The subclass inherits all non-private fields and methods.

super() call

Calls the parent constructor. Must be the FIRST statement. Omitting it inserts super() automatically — only if parent has a no-arg constructor.

protected access

Fields declared protected are accessible in subclasses. Used here so VoitureElectrique can read estDemarree and vitesse directly.

Python comparison

class VoitureElectrique(Voiture): — same concept. But super() must be called explicitly and Java has single inheritance only.

Method Overriding & Polymorphism

```java
// Voiture: standard start
public class Voiture {
    public void demarre() {
        estDemarree = true;
    }
}

// VoitureElectrique: overrides demarre()
public class VoitureElectrique extends Voiture {
    private boolean disjoncteur;

    @Override          // annotation: recommended
    public void demarre() {
        disjoncteur = true;
        super.demarre(); // also run parent's code
    }
}

// Polymorphism in action
List<Voiture> parc = new ArrayList<>();
parc.add(new Voiture(4));
parc.add(new VoitureElectrique(6));
parc.add(new Voiture(28));

for (Voiture v : parc) {
    v.demarre();  // JVM calls the RIGHT version
}
// Voiture.demarre()          for entry 0 and 2
```

@Override annotation

Not required but strongly recommended. Compiler catches typos in the method signature immediately.

Dynamic dispatch

The JVM calls the method of the actual runtime type, not the declared type. List<Voiture> can hold any subclass.

super.method()

Calls the parent version from within a subclass. Useful to extend, not replace, the parent behaviour.

final keyword

public final void demarre() — prevents overriding. final class Voiture — prevents subclassing entirely (e.g. String is final).

Abstract Classes — the VoitureDecapotable Example

```java
// Abstract class: cannot be instantiated
public abstract class VoitureDecapotable
        extends Voiture {

    protected boolean toitReplie;

    public VoitureDecapotable(int p) {
        super(p);
        toitReplie = false;
    }

    // No body — subclass MUST implement
    public abstract void replieLeToit();

    // Shared concrete method
    public boolean estDecapotee() {
        return toitReplie;
    }
}
```

```java
// DeuxChevaux implements replieLeToit()
public class DeuxChevaux
        extends VoitureDecapotable {

    private boolean capoteAttachee;

    public DeuxChevaux(int p) { super(p); }

    @Override
    public void replieLeToit() {
        toitReplie     = true;
        capoteAttachee = true;
    }
}

// C3Pluriel does it differently
public class C3Pluriel
        extends VoitureDecapotable {
    private boolean arceauxRetires;
    public C3Pluriel(int p) { super(p); }
    @Override
```

Cannot instantiate
new VoitureDecapotable() → ERROR

Abstract methods
No body; subclass must override

Concrete methods
Shared implementation (estDecapotee)

Has fields & ctor
And can call super()

Subclass must implement all
Or be abstract itself

Interfaces — Contracts

```
// Interface: a pure contract
// (from your slides: Demarrable)
public interface Demarrable {
    void demarre();      // implicitly public abstract
}

public interface Chargeable {
    void charger(int minutes);
    default String getStatut() {
        return "En charge...";  // default method (Java 
8+)
    }
}

// A class can implement MULTIPLE interfaces
public class VoitureElectrique extends Voiture
        implements Demarrable, Chargeable {

    @Override
    public void demarre() {
        disjoncteur = true;
        estDemarree = true;
    }
    @Override
    public void charger(int minutes) {
        System.out.println("Charging " + minutes + " 
min");
    }
```

Interface rules

All methods public abstract by default. Fields are public static final. One class can implement many interfaces.

Default methods (Java 8+)

interface Chargeable { default String getStatut() {...} } — provides a fallback; implementing classes may override.

Interface as type

Declare Demarrable d = new VoitureElectrique(). Caller only sees the contract — not the implementation.

	Abstract class	Interface
Fields	any	constants only
Constructor	yes	no
Multiple	one (extends)	many (implements)

Programming to Interfaces — Comparable<T>

```
// Voiture implements Comparable → sortable
public class Voiture implements Comparable<Voiture> {
    // ... existing fields ...

    @Override
    public int compareTo(Voiture other) {
        // negative → this < other (less powerful)
        // 0        → equal power
        // positive → this > other (more powerful)
        return Integer.compare(
            this.puissance, other.puissance);
    }
}

// Now you can sort any List<Voiture>
List<Voiture> parc = new ArrayList<>();
parc.add(new Voiture(28));
parc.add(new Voiture(4));
parc.add(new Voiture(6));

Collections.sort(parc);  // uses compareTo()
// [Voiture(4), Voiture(6), Voiture(28)]

// Sort descending with lambda
parc.sort((a, b) ->
    Integer.compare(b.puissance, a.puissance));
```

Comparable<T> contract

compareTo() must be consistent: if a<b then b>a; if a==b then b==a. Return negative, zero, or positive.

Integer.compare()

Never compute a - b for integers (overflow risk). Always use Integer.compare(x,y) or Double.compare(x,y).

Lambda as interface

Any @FunctionalInterface (one abstract method) can be implemented with a lambda. parc.sort((a,b) -> ...) works because Comparator is a FunctionalInterface.

Use case: TreeSet, TreeMap

Add Comparable objects to a TreeSet and they are kept sorted automatically. HashMap uses hashCode() — seen in next slide.

static Fields & Methods

```
public class Voiture {
    // static field: ONE copy shared by ALL instances
    private static int nbVoitCreees = 0;
    public static final int PTAC_MAX = 3500; // constant

    private int puissance;

    public Voiture(int p) {
        puissance = p;
        nbVoitCreees++;  // increments each time
    }

    // static method: no 'this', no instance fields
    public static int getNbVoitCreees() {
        return nbVoitCreees;
    }

    // static factory method (design pattern)
    public static Voiture creerVoitureDefaut() {
        return new Voiture(5);
    }
}

Voiture v1 = new Voiture(4);
Voiture v2 = new Voiture(6);
Voiture v3 = new Voiture(28);
```

```
// Call on the class name, not on an instance
```

static field

One copy per class, shared by all instances. Good for counters, caches, shared config.

static method

No implicit this — cannot access instance fields. Called on the class: Voiture.getNbVoitCreees(). Good for utilities and factory methods.

static final = constant

Convention: ALL_CAPS_WITH_UNDERSCORES. Initialised once, never changed. public static final = global constant.

main() is static — why?

The JVM starts before any object exists. main() must be static so it can be called without creating an instance first.

Inherited from Object — equals() & hashCode()

Every Java class inherits from Object

Method	Return	Default behaviour
`toString()`	String	Returns a text representation — default: ClassName@hashcode
`equals(Object o)`	boolean	Content equality — default: same as == (reference)
`hashCode()`	int	Integer fingerprint of object — used by HashMap, HashSet
`getClass()`	Class<?>	Returns the runtime class object
`clone()`	Object	Creates a copy — requires Cloneable interface

Default (wrong for content comparison)

```
Voiture v1 = new Voiture(5);
Voiture v2 = new Voiture(5);

v1.equals(v2)    // false ← uses ==
                  // (two different objects)
System.out.println(v1);
// Voiture@1a2b3c  ← ugly default
```

Override both — consistent contract

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof Voiture)) return false;
    Voiture other = (Voiture) obj;
    return this.puissance == other.puissance;
}
@Override
public int hashCode() {
    return Integer.hashCode(puissance);
}
// Now: v1.equals(v2) → true
```

Composition vs Inheritance

Inheritance (is-a)

```
// A Car IS-A Vehicle → OK
class Voiture extends Vehicule { ... }

// But: class Voiture extends Moteur ?
// A Voiture IS-A Moteur? NO!
// A Voiture HAS-A Moteur → composition

// Tight coupling problem:
// change parent → may break all subclasses
```

Tight coupling: hard to change parent without breaking subclasses.

Composition (has-a)

```
public class Moteur {
    private int puissance;
    public void start() { ... }
}
public class GPS {
    public String getPosition() { ... }
}

// Voiture HAS-A Moteur, HAS-A GPS
public class Voiture {
    private Moteur moteur; // composition
    private GPS    gps;

    public Voiture(int p) {
        moteur = new Moteur(p);
        gps    = new GPS();
    }
}
```

Rule: prefer composition over inheritance

Use inheritance only for genuine is-a relationships. When in doubt, compose. This is a core principle of the Gang of Four design patterns.

OOP Quick Reference

Concept	Keyword	Key rule	Python equiv.
Inheritance	`extends`	Single class only; super() must be first	`class Dog(Animal)`
Override	`@Override`	Same signature; compiler checks; use super.m() to extend	`def method redef.`
Abstract class	`abstract class`	Cannot instantiate; may have concrete methods & fields	`ABC / @abstractmethod`
Interface	`interface / implements`	No state; class can implement many; use as type	`Protocol (Python 3.8+)`
static	`static`	Class-level; no this; call via ClassName.method()	`class var / @classmethod`
equals/hashCode	`@Override`	Always override both together; required for HashMap/HashSet	`__eq__ / __hash__`

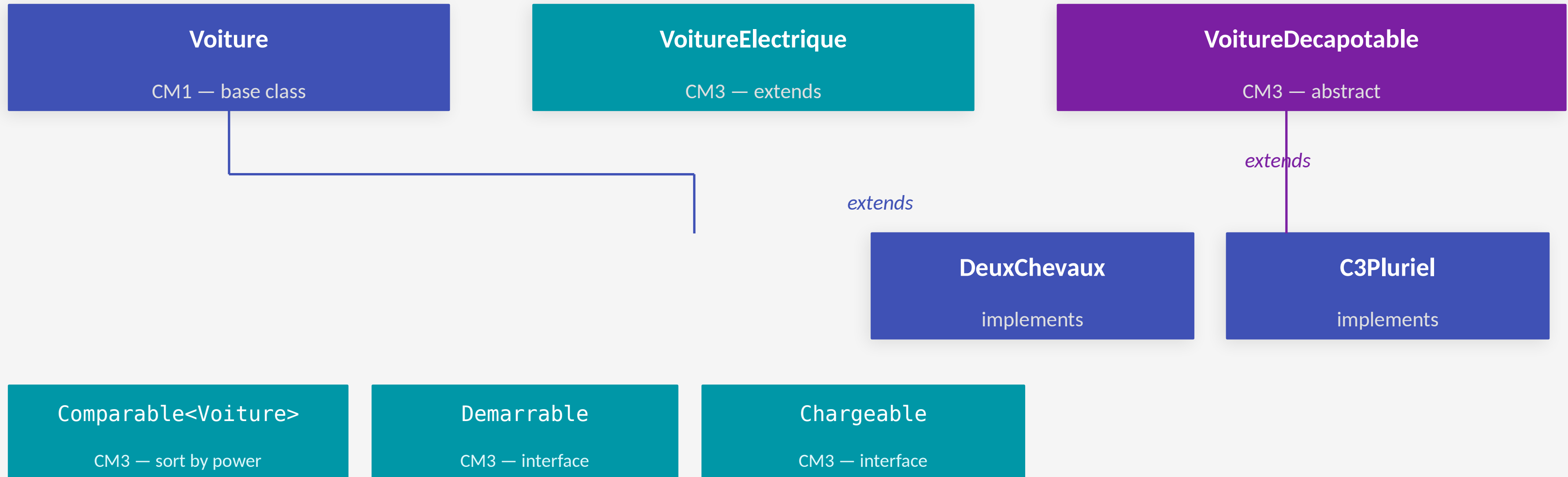
CM3 — Key Takeaways

`extends`	Single inheritance; super() first; protected for subclass-visible members
`@Override`	Dynamic dispatch — JVM picks the right version at runtime
`abstract`	Cannot instantiate; abstract methods force subclasses to implement
`interface`	Pure contract; implement many; declare variables as interface type
`static`	Class-level; constants as public static final ALL_CAPS
`equals/hashCode`	Always override both together — required for collections

Up next → TD3: VoitureDecapotable hierarchy, interfaces & polymorphism

Part 1 Recap — The Voiture Hierarchy

Everything we built in Part 1, together



Encapsulation — private fields, public getters — CM1

Exceptions — InsufficientFundsException, try/catch — CM2

Inheritance — VoitureElectrique, VoitureDecapotable — CM3

Collections — List<Voiture>, Map<String,Voiture> — CM2

File I/O — save/load fleet to .txt — CM2

Interfaces — Comparable, Demarrable, Chargeable — CM3

Beyond This Course — Java Topics Not Covered

The Java language has much more. Here is what we deliberately skipped:

Generics (templates)

class Pair<A,B> { ... } — parameterised classes. Seen implicitly via ArrayList<T>.

JDK docs: java.util.generics

Lambdas & streams

list.stream().filter(v -> v.deQuellePuissance() > 5).collect(...) — functional-style pipelines.

Java 8+ feature

Nested & anonymous classes

new ActionListener() { public void actionPerformed(...) {...} } — seen briefly in Swing.

Partly used in Part 2

Enumerations

enum Day { MON, TUE, WED, ... } — type-safe constants with methods.

java.lang.Enum

Concurrency (threads)

Thread, Runnable, synchronized, ExecutorService — multi-threaded programming.

Out of scope

Records & sealed classes

record Point(int x, int y) {} — immutable data carriers (Java 16+).

Java 16+ feature