1) Concepts of encapsulation, inheritence, and polymorphism

1. The concept of encapsulation in Java allows you to have all the code and data in a one big "box". All variables and methods are defined in one class, in one "box". Encapsulation is about combining methods with data members. With encapsulation you can control acces to data and combining methods with data members. For that, ther are in java 4 levels permissions : public, private, protected and defaults.
2. Setter methods are used to modify privates varibles and getter methods are used to get the data content in variables privates.
3. This is a reference variable wich is used to refer to the current object class instance. This can use attributes of the current class instance to prevent ambiguity between attributes and parameters like in constructor. This can also be used for constructor chaining.
   Super is a reference variable wich is used to refer immediate parent class object. Super can call parent's methods or use parent's variables. For exemple in child constructor, you can call directly parent's constructor with super to initialize instance of child class.
4. The concept of inheritence is when you creat a new class from a class already existing. This classes are the called respectively child class and parent class. In java, inhertitence is depict by the Object class beacause all the others class (like String or Intenger class for exemple) inherit of Object. So when you creat a new class, you can use the differents defaults methods thanks to encapsulation because your new class is automatically "connected" to Object class.
5. The concept of polymorphism allow to a class to have differents types. This is the case when class inherit of a parent class. There are three main kinds of plymorphism :

   . ad-hoc polymorphism (overloading of operators, functions, or methods) when for exemple you have two constructors :

```
class TwoConstructors {
        private int test;

        public TwoConstructors(){
        }
        public TwoConstructors(int test){
                this.test = test;
        }

}
```

. subtype/inclusion polymorphism and parametric polymorphism (Java Generics) :

```
public class Vehicule {
    public int speed;
    public String name;

    public void move(){
        speed++;
    }
    public Vehicule(){

    }

    public Vehicule(String name){
        this.name = name;
    }

}

public class Bike extends Vehicule{

    @Override
    public void move(){
        speed += 3;
    }

    public Bike(){
        super("bike");
    }
}

public class Car extends Vehicule{
    public String name = "car";

    @Override
    public void move(){
        speed+= 2;
    }

    public Car(){
        super("car");
    }
}
public class Main {
```

```
    public static void main(String[] args) {
        Vehicule[] parking = new Vehicule[2];
        parking[0] = new Car();
        parking[1] = new Bike();

        for(Vehicule p : parking){
            p.move();
            System.out.println("name : " + p.name + " speed : " + p.speed);
        }


    }
}
```

6. Like you can see in the previous exemple, sub-type/inclusion polymorphism is linked to inheritence. Indeed, a variable of parent class type can store an element with child type because attributes in parent class are immediatly connected to child class and methods too thanks to Overriding so when variable have parent class type and store a child type element and we don't use respectives attribut and methods of child class but only attributes and methods of super class there is no problems.
7.

```
class HelloEncapsulationTest {
    @Test
    void setPropValTest(){
        HelloEncapsulation test = new HelloEncapsulation(1);
        int test1 = test.getPropVal();
        test.setPropVal(5);
        assertNotEquals(test.getPropVal(), test1);
    }

}
```

```
protected void increaseWealth() {
   fortune = fortune.add(BigDecimal.TEN);
}


//_____________________________________________

class RichDadTest {
    @Test
    void increaseWealthTest(){
```

```
        RichDad richDad = new RichDad("R", "Doe", BigDecimal.valueOf(1),
List.of("School Mate"));
        richDad.increaseWealth();
        assertEquals(BigDecimal.valueOf(11), richDad.getFortune());
    }
}
```

```
@Override
protected void increaseWealth() {
    fortune = fortune.add(BigDecimal.ONE);
}

//_____________________________________________

class RichDadsKidTest {
    @Test
    void increaseWealthTest(){
        RichDadsKid richDadsKid = new RichDadsKid("R", "Doe",
BigDecimal.valueOf(1), List.of("School Mate"));
        richDadsKid.increaseWealth();
        assertEquals(BigDecimal.valueOf(2), richDadsKid.getFortune());
    }
}
```

```
package agh.ii.prinjava.lab01.lst01_03;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class CircleTest {
    @Test
    void circle5PerimeterEaqual10pi(){
        Circle c = new Circle(5);
        assertEquals(10*Math.PI, c.perimeter());
    }
    @Test
    void circle5AreaEaqual25pi(){
        Circle c = new Circle(5);
        assertEquals(25*Math.PI, c.area());
    }
}
```

```java
package agh.ii.prinjava.lab01.lst01_03;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class RectangleTest {
    @Test
    void Rectangle2X5PerimeterEaqual14(){
        Rectangle r = new Rectangle(2, 5);
        assertEquals(14, r.perimeter());
    }
    @Test
    void Rectangle2X5AreaEqual10(){
        Rectangle r = new Rectangle(2, 5);
        assertEquals(10, r.area());
    }
}
```

2) Static members (variables/constants and methods)

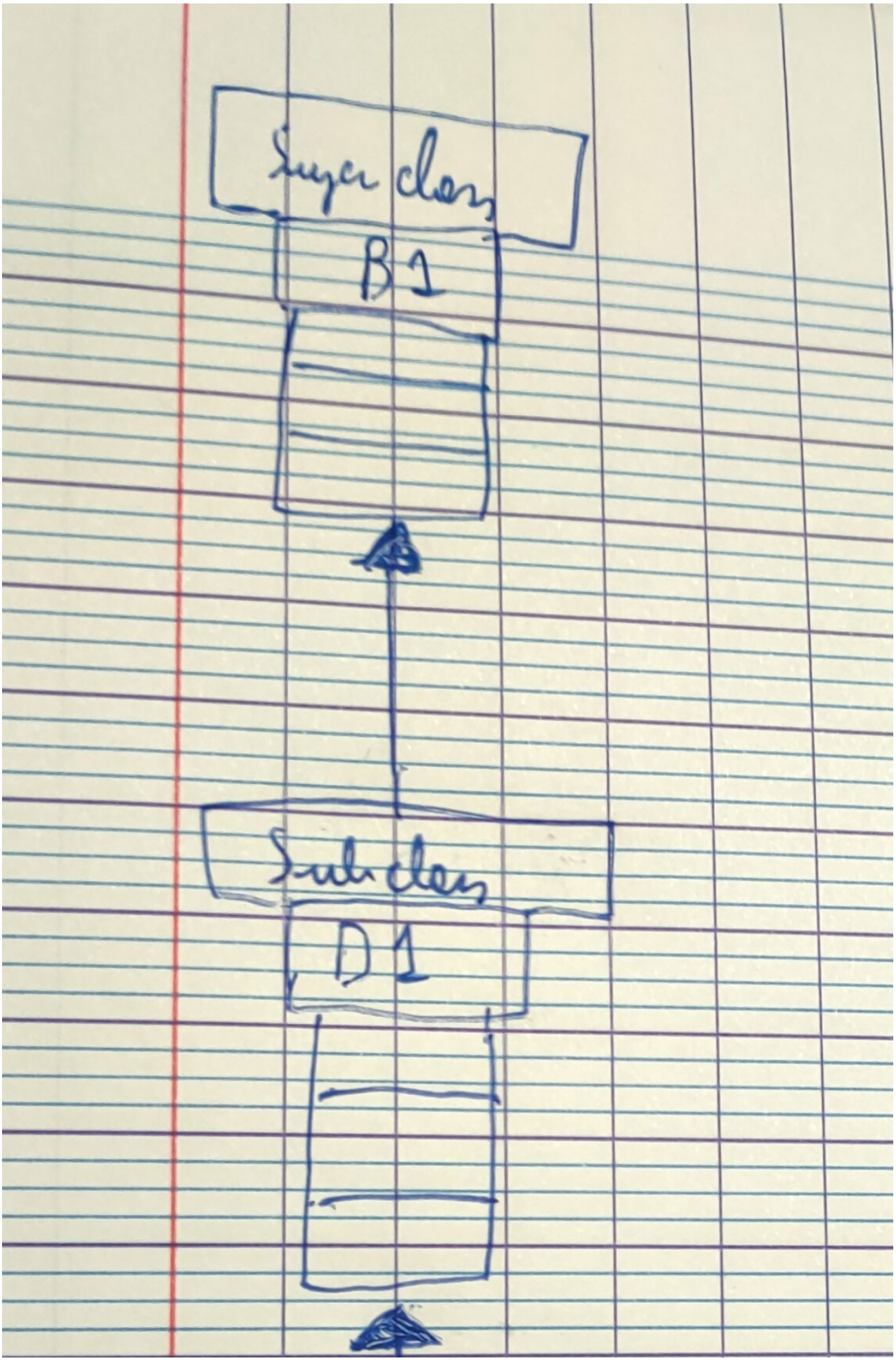
1. . A static variable is directly implemented in the definition of its class. It is therefore never initialized in instances of different objects of this class, even if the different instances of the class will still be able to have access to it. A static variable is therefore a kind of global variable
   . A static constant (static final) is directly implemented in the definition of its class. It is therefore never initialized in instances of different objects of this class. Unlike a static variable, the constant can never be modified.
   . A static method is directly implemented in the definition of its class. We could say that the method adopts the behavior of a global function
2. Because a constant is often useless if you can never access it.
3. Beacause a static method is directly implemented in the definition of its class so when initializing a new object, the static method is absolutely not "connected to it but remains within the class
4.

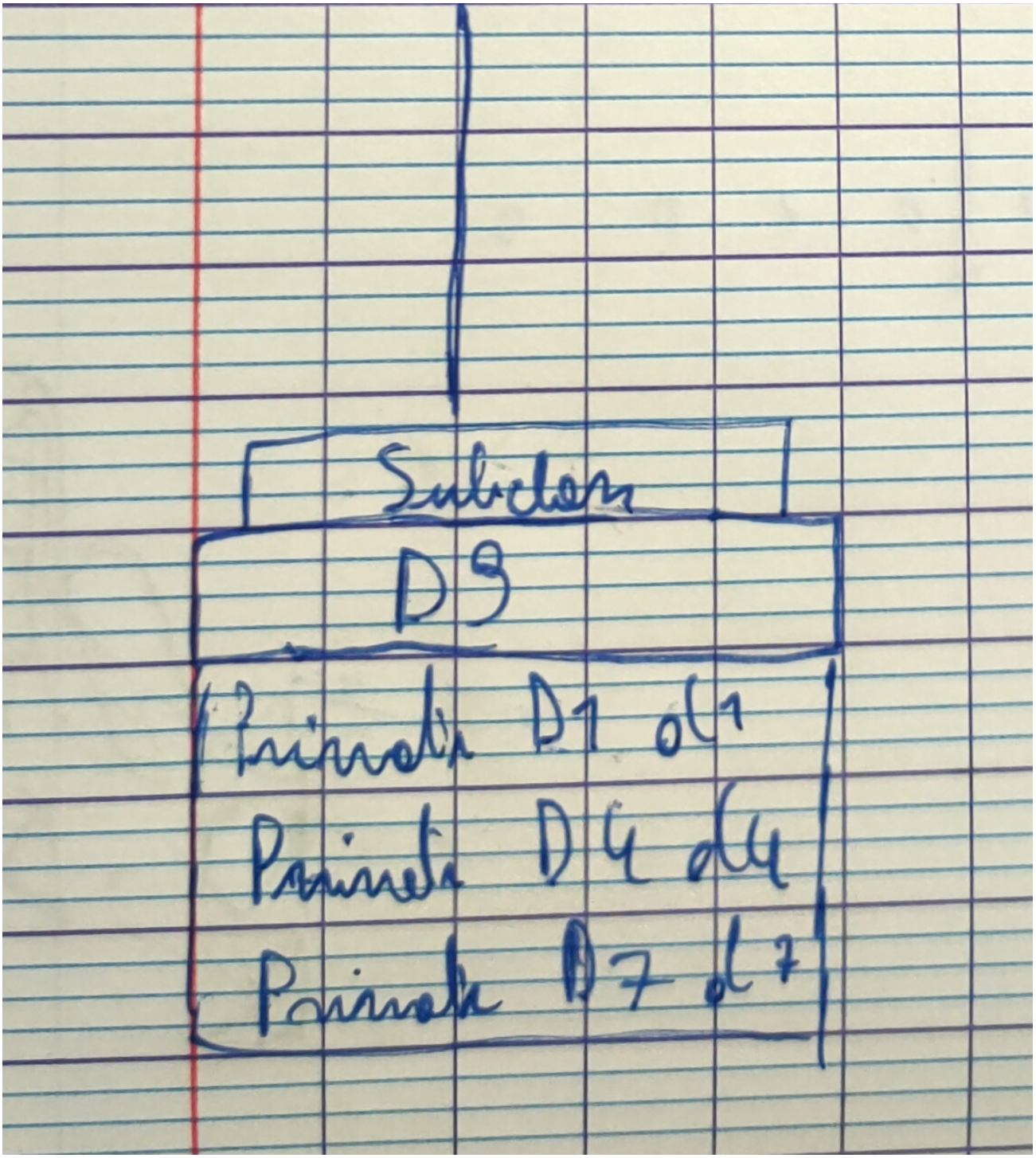
```java
public class Bank {
    public String name;
    private static int money;

    public static int getMoney(){
        return money;
```

```
    }
}

public class Main {

        Bank.getMoney();

    }
}
```

3) Constructors, factory methods, and singletons

1. The process of initializing an object begins with the storage of variables (whether constant or static). Then there are the methods (static or not). Then there are the anonymous static blocks, followed by the anonymous blocks only if the class is loaded into memory, since if it were just an instance initialization, only the anonymous blocks would be involved.
   Finally, we call the various constructors according to the given parameters, which will finish the process by loading the attributes.
2. 1-Diagram :


Super class
B1
Sub class
D1



The first constructor called is D7() {} because of the line "private D7 d7 = new D7();" then the D7() constructor calls D7(int x) which then calls B3(int x). Then there's the anonymous block with d4 = new D4(); which calls the D4() constructor, which in turn calls the B3(int x) constructor.
Finally, the D9() constructor is called, which in turn calls the D1() constructor with super(); and re-calls it with d1 = new D1();

3. In Java, the constructor initializes a new instance of a class with the keyword new. The constructor has the same name as its class, has no return type and must be public if it is to

be used. The factory methods, create and buildFrom, on the other hand, are static methods (i.e. they are used with the class definition) and return the name of their class as the return type. In a way, they can return the initialization of the class instance with the constructor, using "return new ClassName();".
You could say that factory methods work a bit like the getter method of a private variable with a private constructor.

4.

```
public class Game {
    private static volatile Game game = null;
    public String name;
    public int points;
    public int playerNumber;
    private Game(){
    }
    public static Game getInstance(){
        if(game == null){
            synchronized (Game.class){
                if(game == null){
                    game = new Game();
                }
            }
        }
        return game;
    }
}
```

```
public class KingOfPoland {
    private static KingOfPoland king = null;
    public int age;
    public String name;
    public String title;
    private KingOfPoland(){
    }
    public static KingOfPoland getInstance(){
        if(king == null){
            synchronized (KingOfPoland.class){
                if(king == null){
                    king = new KingOfPoland();
                }
            }
        }
        return king;
```

```
    }
}
```

5.

```
 class EagerSingletonTest {
    @Test
    void getInstanceTest(){
        EagerSingleton e1 = EagerSingleton.getInstance();
        EagerSingleton e2 = EagerSingleton.getInstance();
        assertSame(e1, e2);
    }
}
```

```
class UnsafeSingletonTest {
    @Test
    void getInstanceTest(){
        UnsafeSingleton e1 = UnsafeSingleton.getInstance();
        UnsafeSingleton e2 = UnsafeSingleton.getInstance();
        assertEquals(e1, e2);
    }
}
```

```
class LazySingletonTest {
    @Test
    void getInstanceTest(){
        LazySingleton e1 = LazySingleton.getInstance();
        LazySingleton e2 = LazySingleton.getInstance();
        assertTrue(e1 == e2);
    }
}
```

4) Immutable objects/classes and Java Records

1. The strategy for defining an immutable object is to first define the class as a final class (a constant class) to remove any ambiguity with possible subclasses. Next, set the constructor to private and give preference to factory variables. Then set all possible fields (variables and methods) to final and private. Finally, delete all setters
2. A class is immutable if the state of its instances cannot change after their creation.
3. Immutable objects are certified thread-safe. They need neither a copy constructor nor an interface implementation. There is an absence of hidden side-effects and protection

against null reference errors. It also brings a prevention of identity mutation.

4. The use of records facilitates rapid understanding of the role of the defined type. Example:

```
public record Person(String name, String adresse, int age) {
}

public record Employee(Person person, float salary, String compagny) {
}
```

5.

```
class HelloImmutableTest {
    @Test
    void testHelloImmutableTestIsEqual(){
        HelloImmutable test = new HelloImmutable (1, "test");
        HelloImmutable test1 = new HelloImmutable (1, "test");
        assertTrue(test.equals(test1));
    }

    @Test
    void testHelloImmutableTestIsHashSame(){
        HelloImmutable test = new HelloImmutable (1, "test");
        HelloImmutable test1 = new HelloImmutable (1, "test");
        assertEquals(test.hashCode(), test1.hashCode());
    }

    @Test
    void testHelloImmutableTestIstoStringIsSame(){
        HelloImmutable test = new HelloImmutable (1, "test");
        HelloImmutable test1 = new HelloImmutable (1, "test");
        assertEquals(test.toString(), test1.toString());
    }
}
```

```
class HelloJavaRecordTest {
    @Test
    void testHelloJavaRecordTestIsEquals(){
        HelloJavaRecord test = new HelloJavaRecord(2, "test2");
        HelloJavaRecord test1 = new HelloJavaRecord(2, "test2");
        assertTrue(test.equals(test1));
    }

    @Test
    void testHelloJavaRecordTestIsHashSame() {
```

```
        HelloJavaRecord test = new HelloJavaRecord(2, "test2");
        HelloJavaRecord test1 = new HelloJavaRecord(2, "test2");
        assertEquals(test.hashCode(), test1.hashCode());
    }

    @Test
    void testHelloJavaRecordTestIstoStringIsSame() {
        HelloJavaRecord test = new HelloJavaRecord(2, "test2");
        HelloJavaRecord test1 = new HelloJavaRecord(2, "test2");
        assertEquals(test.toString(), test1.toString());
    }

}
```

5) Overriding hashCode, equals, and toString

1. In java, == sert à savoir si 2 objets ont la même référence (si les deux variables "pointent" vers le même objet). The equals method, on the other hand, can be used to determine the equality of two objects by comparing, for example, the respective equality of all their attributes. Exemple :

```
public class Exemple {
    int a, b;
    String c;
    public Exemple(int a, int b, String c){
        this.a = a;
        this.b = b;
        this.c = c;
    }
    @Override
    public boolean equals(Object o){
        if(o == this){
            return true;
        }
        if(!(o instanceof Exemple e)){
            return false;
        }

        //Exemple e = (Exemple) o;
        return a==e.a && b==e.b && c.equals(e.c);
    }
}

//------------------------------------------------------------------------
```

```
public class Main {

    public static void main(String[] args) {

    Exemple e1 = new Exemple(1,2,"aa");
    Exemple e2 = e1;
    Exemple e3 = new Exemple(1, 2, "aa");

    System.out.println(e1.equals(e3));
    System.out.println(e1 == e2);

    }
}
```

2. A hash code is an integer value that is associated with each object in Java so we can say that it's its identity, just as we can say that the toString method is too, but in a different way. So if two object are equal then they have the same hash code.
3. Whenever the equals method is overloaded, it is necessary to overload the hashCode method, in order to maintain the general contract for the hashCode method, which stipulates that equal objects must have equal hash codes.