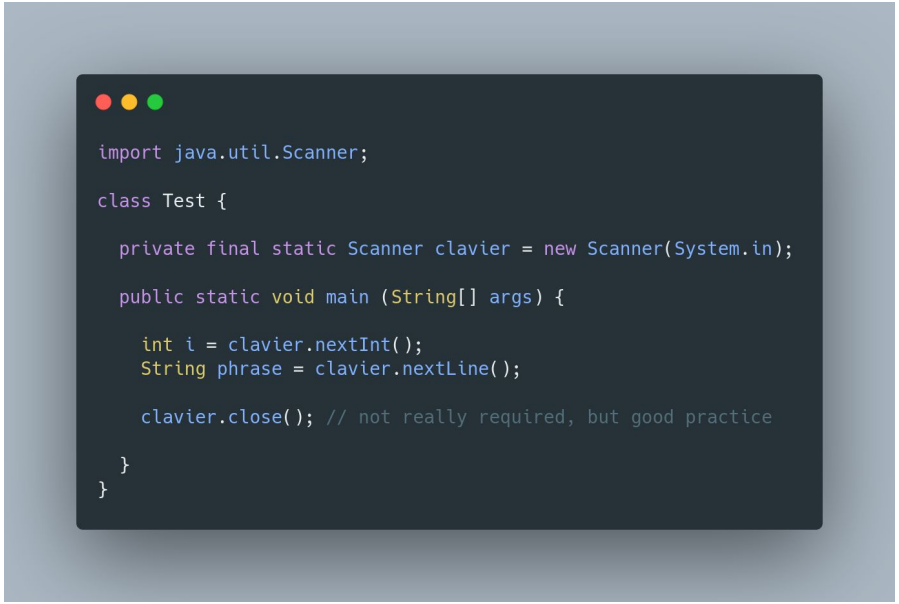
Exam de prog

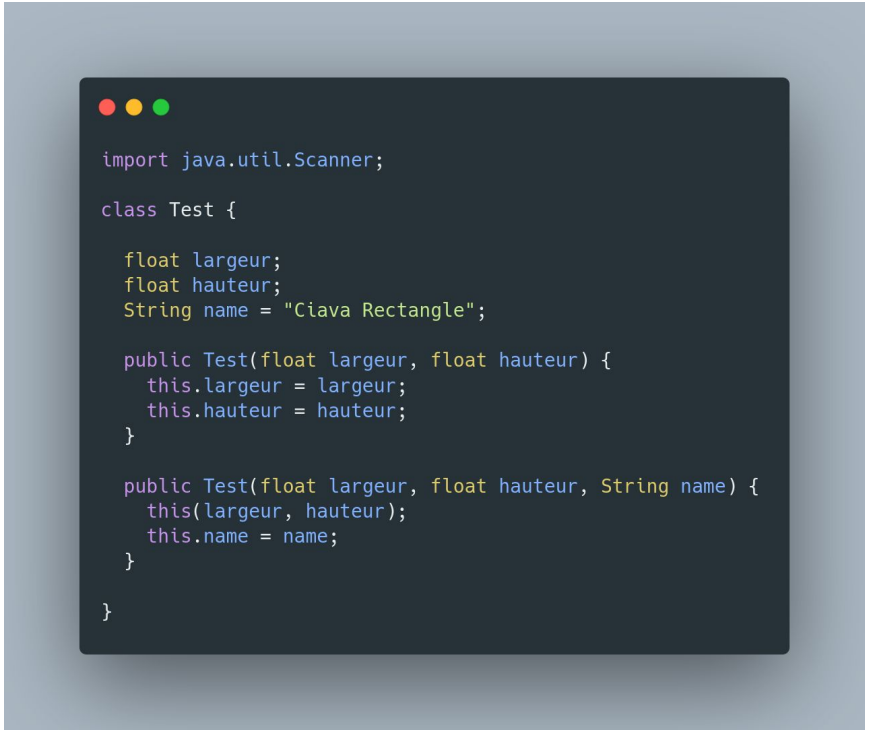
Java Scanner (exemples)



```java
import java.util.Scanner;

class Test {

  private final static Scanner clavier = new Scanner(System.in);

  public static void main (String[] args) {

    int i = clavier.nextInt();
    String phrase = clavier.nextLine();

    clavier.close(); // not really required, but good practice

  }
}
```

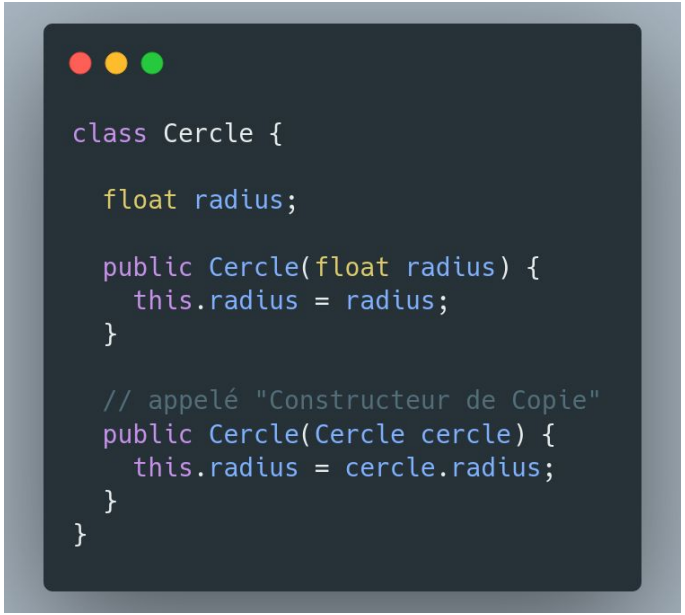
Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

Constructeurs (exemples)



```java
import java.util.Scanner;

class Test {

  float largeur;
  float hauteur;
  String name = "Ciava Rectangle";

  public Test(float largeur, float hauteur) {
    this.largeur = largeur;
    this.hauteur = hauteur;
  }

  public Test(float largeur, float hauteur, String name) {
    this(largeur, hauteur);
    this.name = name;
  }

}
```

attention: ne pas dupliquer les constructeurs, utiliser this() à la place si on a plus de paramètres dans certains cas.



```java
class Cercle {

  float radius;

  public Cercle(float radius) {
    this.radius = radius;
  }

  // appelé "Constructeur de Copie"
  public Cercle(Cercle cercle) {
    this.radius = cercle.radius;
  }
}
```

attention: en java il n'y a pas de constructeur de copie par défaut généré automatiquement

Constructeurs (QCM)

- Si aucun constructeur par défaut n'est spécifié, le compilateur va automatiquement en générer un.
- Si aucun constructeur par défaut n'est spécifié, le compilateur va à la place appeler la méthode init() de la classe.
- Les constructeurs ont toujours le type ***void*** comme type de retour.
- Le constructeur par défaut n'accepte pas de paramètre.
- Lorsque plusieurs constructeurs sont définis pour une même classe, ils seront exécutés dans l'ordre de définition.
- Les constructeurs doivent avoir le même nom que la classe.
- Il est possible de ne définir qu'un seul constructeur par classe.
- Il est uniquement possible de définir un seul constructeur par classe.
- Si aucune constructeur par défaut n'est défini, il n'est pas possible de créer une instance de la classe sans paramètre.
- Le constructeur par défaut est le seul à avoir le même nom que la classe.
- Il est nécessaire d'explicitement définir au moins un constructeur, avec ou sans paramètres.
- Lorsque plusieurs constructeurs sont définis pour une classe, le compilateur en choisit un au hasard pour construire les objets
- Le constructeur par défaut n'est pas appelé si il existe déjà une instance de la classe.
- Les méthodes d'une classe ne peuvent pas être appelée à partir des constructeurs car elles ne sont pas encore initialisées.
- Le constructeur par défaut est automatiquement appelé si une instance de la classe est créée sans fournir de paramètre.
- Il est nécessaire de définir au moins un constructeur pour chaque classe.
- Il n'est pas possible de définir un constructeur sans paramètres car il est déjà défini automatiquement par le compilateur.
- En java il n'y a pas de constructeur de copie par défaut généré automatiquement

Héritage (QCM + exemples)

- La classe parente connaît toutes ses sous-classes.
- La classe parente a accès à tous les membres de ses sous-classes.
- Une sous-classe a uniquement accès aux membres de sa super-classe directe (et donc pas à la super-classe de la super-classe)
- La classe parente ne peut pas définir de membres privés.
- L'héritage définit une relation « A-UN/POSSEDE-UN ».
- L'héritage définit une relation « EST-UN ».
- Une classe ne peut étendre qu'une seule classe.
- Une sous-classe hérite du type de sa super-classe.
- Une classe peut définir plusieurs classes parentes.
- Quand on redéfinit une méthode, il ne faut pas restreindre le droit d'accès comparé à la super classe
- Quand redéfinit une méthode (**et pas une surcharge)**, le type de retour doit être compatible (égal ou plus précis)

Modifier @Override ⇔ redéfinition
≠ surcharge (type des paramètres différents)

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

A someD = new D();
A someB = new B();
```

les B, C, D héritent tous du **type de A**

```
class Region {}

class Ocean extends Region {}

Ocean pacifique = new Ocean();
pacifique instanceof Region; // true
pacifique.getClass().equals(Ocean.class); // true
pacifique.getClass().equals(Region.class); // false
```

```
class Dog {
  public void eat() {
    // laper l'eau
  }
}

class BabyDog extends Dog {
  @Override
  public void eat() {
    // boire le lait
  }
}
```

Héritage (exemples, attributs)

```
ClassC c = new ClassC();
c.print(); // C: 20

ClassB b = new ClassB();
b.print(); // B: 20

ClassA ab = new ClassB();
ab.print(); // B: 20

((ClassB)c).print(); // C: 20
```

Le choix des méthodes dépend du type **réel** de l'objet (B ou C), tandis que les attributs dépendent du type de **référence**.

Pour afficher 10, il faudrait déplacer **print()** dans la classe B.

```
class Animal {
  int gourmandise = 0;
}

class Chien extends Animal {
  int gourmandise = 10;
}

class Test {
  public static void main(String[] args) {
    Chien chien = new Chien();
    System.out.println(chien.gourmandise); // 10
    System.out.println(((Animal) chien).gourmandise); // 0
  }
}
```

```
class ClassA {
    protected int number;

    public ClassA() {
        number = 20;
    }

    public void print() {
        System.out.println(getPrefix() + ": " + number);
    }

    protected String getPrefix() {
        return "A";
    }
}

class ClassB extends ClassA {
    protected int number = 10;

    protected String getPrefix() {
        return "B";
    }
}

class ClassC extends ClassB {
    public ClassC() {
        super();
    }

    protected String getPrefix(){
        return "C";
    }
}
```

Polymorphisme (classes abstraites)

```
abstract class Animal {
  public abstract void eat();
  protected abstract void die();
}

class Chien extends Animal {
  public void eat() {
    // let's go laper l'eau
  }
  protected void die() {
    // bye bye
  }
}

class Cat extends Animal {
  private void eat() {
    // throws an error!
    // we said that all animals
    // had to expose a public eat()
  }
  public void die() {
    // more rights is ok
    // less is not possible
  }
}
```

'die()' in 'Chien' clashes with 'die()' in 'Animal'; attempting to assign weaker access privileges ('private'); was 'protected'

Polymorphisme (QCM classes abstraites)

- Une méthode ne peut pas être abstraite et protégée ("protected") en même temps.
- Les classes abstraites ne peuvent pas définir de constructeurs.
- Une classe abstraite ne peut pas hériter d'une classe non-abstraite.
- Une classe qui hérite d'une classe abstraite doit obligatoirement définir toutes les méthodes abstraites de cette super-classe abstraite.
- Il n'est pas nécessaire qu'une classe abstraite ait des méthodes abstraites.
- On ne peut pas hériter de classes abstraites
- Les classes abstraites ne peuvent être instanciées.
- Une sous-classe qui étend une classe abstraite peut également être définie comme abstraite.
- Les méthodes abstraites ne peuvent pas être appelées dans les classes qui les définissent.
- Une classe non-abstraite ne peut pas hériter d'une classe abstraite.

Polymorphisme (modifier final)

- Seuls les membres publiques peuvent être définis comme **finaux**.
- Un object avec une référence **finale** ne peut pas être modifié.
- Les variables **finales** ne peuvent pas être définies plus d'une fois.
- Seuls les membres privés peuvent être définis comme **finaux**.
- Les méthodes **finales** ne peuvent pas être redéfinies dans les sous-classes.
- Les classes **finales** ne peuvent pas être héritées
- Une classe qui définit des méthodes **finales** doit aussi être déclarée comme **finale**.
- Une méthode **finale** ne peut pas être appelée dans les sous-classes.
- Une classe définissant une méthode **finale** ne peut pas être héritée.
- Une classe **finale** ne peut être instanciée.
- Les variables **finales** doivent commencer par une majuscule.
- Une méthode peut avoir des paramètres **finaux**.

remarque: une constante de classe (donc non propre à l'instance) est normalement toujours final static

Polymorphisme (modifier static)

- Les variables statiques ne peuvent être utilisée que dans des méthodes statiques
- Seules les méthodes peuvent être statiques
- Les variables statiques sont définies dans les méthodes statiques
- Les variables statiques peuvent être utilisées dans des méthodes d'instances

```java
class Rectangle {
  float largeur;
  float hauteur;

  public Rectangle (float largeur, float hauteur) {
    this.largeur = largeur;
    this.hauteur = hauteur;
  }
  @Override
  public boolean equals(Object someObject) {
    if (someObject == null) {
      return false;
    }
    // here we should not use instanceof
    // "convention" is that an object is really equals to another
    // only if it's at the same level
    // Avatar != Elfe even if one extends another
    // and they have the same attributes
    else if (someObject.getClass() !== getClass()) {
      return false;
    } else {
      Rectangle r = (Rectangle) someObject;
      return largeur == r.largeur && hauteur = r.hauteur;
    }
  }
}
```

attention : pas de résolution dynamique des liens quand on utilise le mot-clef static

Méthode **equals()** par défaut compare uniquement les références.

```java
class A {
    public static void print(){
        System.out.println("A");
    }
}

class B extends A {
    public static void print(){
        System.out.println("B");
    }
}

class C extends B {
    public static void print(){
        System.out.println("C");
    }
}

A.print(); // A
B.print(); // B
C.print(); // C

A aInstance1 = new A();
A aInstance2 = new B();
A aInstance3 = new C();

aInstance1.print(); // A
aInstance2.print(); // A
aInstance3.print(); // A
```

Polymorphisme (interfaces)

- Une classe n'a pas le droit d'implémenter deux interfaces qui ont la même méthode
- Une classe peut implémenter plusieurs interfaces
- Une classe qui ne définit que certaines méthodes d'une interface (dépourvue de méthodes avec définition par défaut) qu'elle implémente doit être abstraite
- Une interface doit contenir au moins un constructeur

Tout attribut d'une interface est par défaut **public static abstract final**

Toutes méthodes d'une interface est nécessairement **public** et elle est **abstraite** (sauf s'il existe une déf. par défaut)

```java
interface I1 {
    // implicit definition of variables:
    // public, static, final
    // therefore public is useless here
    public int a = 10;
    default void print() {
        System.out.println("one");
    }
}

interface I2 {
    int a = 20;
    int b = 40;
    default void print() {
        System.out.println("two");
    }
}

class C1 implements I1, I2 {
    int var1;
    public C1(){
        // does not work, it's ambiguious
        var1 = a + b;
        // correct code:
        var1 = I1.a + b;
    }

    // we need an explicit definition
    // of the "right" print
    public void print() {
      I1.super.print();
    }
}
```

Polymorphisme (classes imbriquées)

```
class AChien {

    private String name;
    private PatteDeChien[] pattes = new PatteDeChien[4];

    public AChien(String name) {
        this.name = name;
        this.pattes[0] = new PatteDeChien();
        this.pattes[1] = new PatteDeChien();
        this.pattes[2] = new PatteDeChien();
        this.pattes[3] = new PatteDeChien();

        this.pattes[0].afficheStatutBoitage();
    }

    private class PatteDeChien {

        private void afficheStatutBoitage () {
            System.out.println("Patte de " + AChien.this.name + ": boitage OK");
        }

    }
}

class Test {
    public static void main(String[] args) {
        AChien chien = new AChien("Francis");
    }
}
```

attention: il faut utiliser SuperClass.this pour accéder à l'instance courante dans une classe imbriquée.

lorsque la classe interne n'a pas besoin d'accéder à l'instance courante de la classe externe, elle peut être déclarée comme statique.

si on veut accéder à la classe imbriquée à l'extérieur (via AChien.PatteDeChien), on peut la déclarer comme public **mais on ne pourra pas l'instancier.**

Paquetages

```java
package bank;

class Bank {

  protected void printMoney() {
    System.out.println(666);
  }

}
```

attention : même si on fait p1.p2, p1 et p1.p2 sont deux paquetages complètement séparés. donc les méthodes **protected** ne fonctionnent pas entre eux

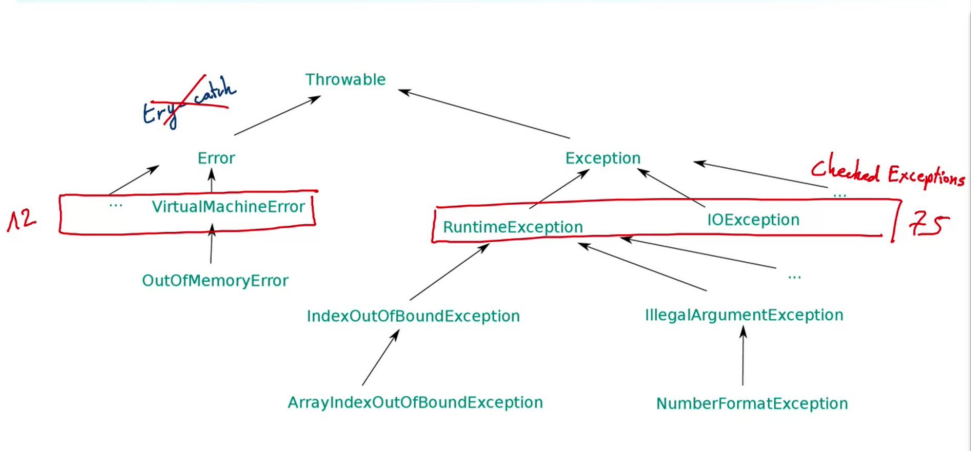
```java
package bank.administration;
import bank.Bank;

class BankAdministration {

  public BankAdministration (Bank bank) {
    bank.printMoney(); // throws an error
    // bank.administration is not the same package
    // as bank
  }

}
```

```java
// Math has a static sin class
// we can import it directly
// using import static
import static java.lang.Math.sin;
```

note : le droit d'accès par défaut d'une méthode d'une classe est private protected (c'est comme protected mais les sous-classes ne peuvent pas y accéder).

Gestion des exceptions

- Une exception peut être traitée à plusieurs endroits différents dans le code.
- un bloc **try** bloc doit être suivi de un ou plusieurs **catch** ou/et d'un bloc **finally**
- Après les blocs **try/catch** block, le bloc **finally** est obligatoire
- Les blocs **try** et **catch** peuvent eux-même lancer des exceptions
- Les exceptions ne sont pas des objets
- **try** permet de lancer une exception
- Le lancement d'une exception (**throw**) et son traitement sont le plus souvent au même endroit dans un programme
- Chaque bloc **try** doit avoir exactement un bloc **catch** correspondant



Throwable
→ Error (12 enfants, **non-checked**)
→ Exception (75 enfants dont 74 **checked** et 1 **non-checked** (RunTimeException)

Syntaxe Basique de Java (Arrays)

```java
// for loop
for (int i = 0; i <= cars.size; i++) {
  System.out.println(cars[i]);
}

// foreach
for (Car car : cars) {
  System.out.println(car);
}
```

attention: foreach est recommandé si vous n'avez pas besoin de l'index

```java
// if you don't know the initial size
ArrayList<Car> scores = new ArrayList<Car>();

Car myCar = new Car();

scores.add(myCar);
// remove can take an int (the index)
// or directly an object to remove
scores.remove(0);
scores.add(myCar);
scores.remove(myCar);
// size of scores? = 0
```

```java
// create a new array
int[] scores = new int[2];
// this also works: (but not recommended)
int scores[] = new int[2];

// you can also not specify the length
// but then you have to specify the elements
int[] scores = new int[]{4,5};
```

Questions tricky

```java
String nomDeMonChat = "Francis";
String nomDuChatDeMonVoisin = nomDeMonChat + "";

nomDeMonChat == nomDuChatDeMonVoisin; // false
nomDeMonChat.equals(nomDuChatDeMonVoisin); // true

String nomDunAutreChat = "Bedou";
String autreNom = "Bedou";

nomDunAutreChat == autreNom; // ... true
```

attention: le == vérifie si les String font référence à la même zone mémoire, ce qui n'est pas le cas dans l'exemple 1.
(mais par optimisation Java créé un pool des String ayant la même valeur donc dans le deuxième exemple elles auront en fait la même adresse)

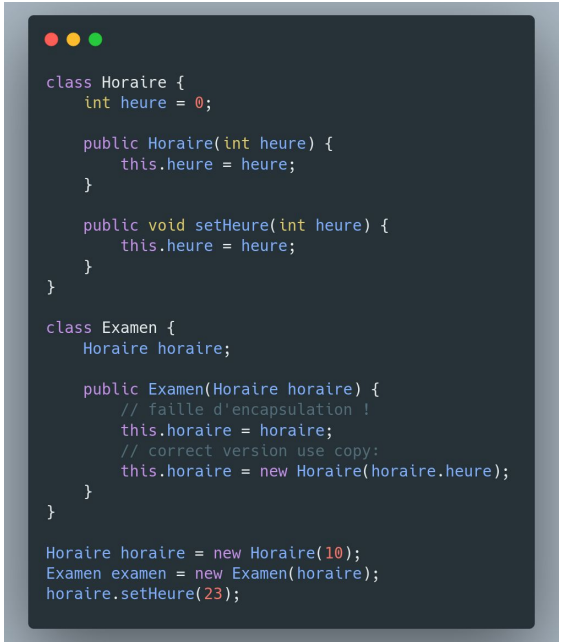
Quand dans les exams ils parlent des « attributs » d'une instance, ils parlent des attributs de l'instance **+** ceux des super classes

Faire attention print ≠ println dans les exos d'analyse de déroulement

---

```java
enum SemaphoreColor {

    RED("rouge", false),
    GREEN("vert", true),
    YELLOW("jaune", false);

    private SemaphoreColor(String name, boolean canPass) {
        this.name = name;
        this.canPass = canPass;
    }

    private String name;
    private boolean canPass;

    public String frenchName() { return name; }
    public boolean canPass() { return canPass; }

}
```

Failles d’encapsulation



```
class Horaire {
    int heure = 0;

    public Horaire(int heure) {
        this.heure = heure;
    }

    public void setHeure(int heure) {
        this.heure = heure;
    }
}

class Examen {
    Horaire horaire;

    public Examen(Horaire horaire) {
        // faille d'encapsulation !
        this.horaire = horaire;
        // correct version use copy:
        this.horaire = new Horaire(horaire.heure);
    }
}

Horaire horaire = new Horaire(10);
Examen examen = new Examen(horaire);
horaire.setHeure(23);
```

en réalité il n’est pas vraiment possible de tout encapsuler. Il faudrait :

→ faire ***a*** = *super.clone()* pour récupérer une instance clonée de la sous-classe actuelle
(oui c’est super bizarre que faire super.clone() renvoie une instance de la sous-classe et pas de la super classe (comment la super classe qui exécute le clone() saurait qu’il faut renvoyer une instance de la sous-classe ? mais bon c’est comme ça que ça marche)

→ assigner à **a** chaque attribut complexes avec **a** = this.attr.clone();
(donc ça nécessite que chaque attribut implémente Cloneable)

→ return **a;** (la sous-classe clonée)

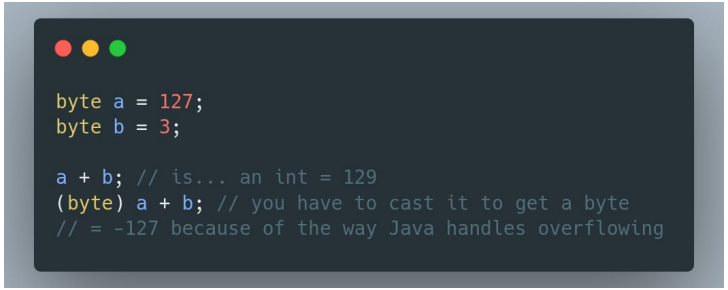
=> faire tout ce travail de copie augmente la complexité et le risque d’erreurs, ça peut déclencher des CloneNotSupportedException qu’il faut gérer partout, clone() créé des objets sans passer par les constructeurs ça peut amener à des résultats incohérents, Cloneable ne force même pas la redéfinition de clone()..

Classe **Mutable** : contient des setters publiques (comme Horaire dans l’exemple) → soumise aux failles d’encapsulations

Classe **Immutable** : pas de setters publiques

solution pas trop compliquée : les getters réalisent une copie défensive (*getHoraire()* pourrait faire *return new Horaire(horaire)*)

Opérateurs binaires (MP1 2023)



```java
byte a = 127;
byte b = 3;

a + b; // is... an int = 129
(byte) a + b; // you have to cast it to get a byte
// = -127 because of the way Java handles overflowing
```

promotion entière : les valeurs plus petites qu'un int sont converties en int puis additionnées ensuites (pour éviter des erreurs de précisions)
donc tjrs faire attention au type de retour que le compilateur peut modifier

- un int est représenté sur 32 bits (4 octets)
- caster un int en byte supprime les 3 octets les plus à gauche
- caster un byte en int ajoute 3 octets à gauche avec des 1 ou des 0 en fonction de si le byte est négatif

```java
byte b = 127;          // 0b01111111
int  i = 127 + 1;      // 0b000000000..10000000 =  128
byte b2 = (byte) i ;   // 0b000000000..10000000 = -128
int  i2 = (byte)-128 ;// 0b111111111_11111111_11111111_10000000
```

Opérateurs binaires (MP1 2023)

Opérateurs << et |

```
int val1 = 0b10;      (2 in decimal)
int val2 = 0b10001;   (17 in decimal)
```

Bitwise-OR pour 1 bit

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

But: `0b10010001`

- Étape 1: shift left << (insertion de 0s à droite)

```
int val1_sl = val1 << 6;
```

résultat binaire: `10000000` (2 • $2^6 = 2^7$ = 128 in decimal)

Étape 2: bitwise-OR |

```
int val3 = val1_sl | val2;
```

résultat binaire: `10010001`
(128 + 17 = 145 in decimal)

val1_s1 | val2

1	0	0	0	0	0	0	0
			1	0	0	0	1
1	0	0	1	0	0	0	1

Opérateurs >>>/>> et &

```
int val1 = 0b10010001;
```

- Extraction des bits "rouge" : shift right* >>>

```
int val2 = val1 >>> 6;
```

Output: `0b10`

Extraction des bits "bleus": bitwise-and (ou masque) &

```
int val3 = val1 & 0b111111;
```

Output: `0b10010001`
`0b010001`

Bitwise-AND pour 1 bi

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

val1 & 0x111111

1	0	0	1	0	0	0	1
			1	1	1	1	1
0	0	0	1	0	0	0	1

* Deux versions: >> and >>>:
  >>> insertion de 0s à gauche
  >> insertion de 0s pour les nombres positifs et de 1s pour les négatifs sur la gauche

Exercice de conception

- vérifier que les class implémentent bien “Updatable” ou “Drawable”
- vérifier le type de update()
- utiliser List au lieu de ArrayList (“code an interface not an implementation”)
- redéfinir les fonctions et mettre des @Override
- ne pas oublier de mettre des private, public, final static