1 Bases et Pièges Classiques

1.1 Comparaison

— **Primitifs** (`int`, `double`, `boolean`) : Utiliser `==`.

— **Objets** (`String`, `Tableaux`, `Instances`) :

  — `a == b` : Compare les **références** (adresses mémoire).

  — `a.equals(b)` : Compare le **contenu** (si redéfini).

— **String Pool :** `"A" == "A"` (true), mais `new String("A")== "A"` (false). Toujours utiliser `.equals()` pour les Strings !

1.2 Tableaux vs ArrayList

Tableau (Fixe)	**ArrayList (Dynamique)**
`int[] t = new int [5];`	`ArrayList<Integer> l = ... ;`
`t.length` (attribut)	`l.size()` (méthode)
`t[i]`	`l.get(i)`
`t[i] = val`	`l.set(i, val)`
Immuable en taille	`l.add(val)`, `l.remove(i)`

TABLE 1 – Tableau de comparaison entre les liste de taille fixe et dynamique

2 Programmation Orientée Objet

2.1 Constructeurs

— **Par défaut :** Fourni par Java *si aucun autre constructeur n'existe*. Initialise à `0`/`null`/`false`.

— **Chainage (this)** : `this(...)` appelle un autre constructeur de la **même classe**. Doit être la **1ère ligne**.

— **Chainage (super)** : `super(...)` appelle le constructeur de la **super-classe**. C'est la manifestation du **polymorphisme en construction** : l'objet se construit "de la base vers le dérivé".

— **Piège de la résolution dynamique** : Si un constructeur de la super-classe appelle une méthode qui est redéfinie dans la sous-classe, c'est la version de la **sous-classe** qui est exécutée, *même si le constructeur de la sous-classe n'a pas encore été appelé*. Attention aux `NullPointerException` si la méthode redéfinie utilise des champs initialisés uniquement dans le constructeur fils.

— **Règle d'or** : Si la 1ère ligne n'est pas `this(...)` ou `super(...)`, Java ajoute implicitement `super()` (vide). Si le parent n'a pas de constructeur par défaut → Erreur de compilation.

— **Récursivité** : Un constructeur **ne peut pas** s'appeler lui-même (ni directement, ni indirectement). Le compilateur l'interdit pour éviter une boucle d'initialisation infinie. Il peut cependant appeler une méthode externe qui est, elle, récursive.

2.2 Encapsulation & Visibilité

Modifier	Classe	Pkg	Sous-Cl	Monde
`public`	Oui	Oui	Oui	Oui
`protected`	Oui	Oui	Oui	Non
`(défaut)`	Oui	Oui	Non	Non
`private`	Oui	Non	Non	Non

TABLE 2 – Tableau de visibilité des modificateurs

Lors de la redéfinition d'une méthode héritée, il est **interdit de réduire la visibilité** de la méthode par rapport à sa version dans la super-classe. La visibilité peut être maintenue ou étendue (ex : de `protected` à `public`), mais jamais restreinte (ex : de `public` à `protected` ou `private`).

2.3 Static vs Final

- `static` : Appartient à la classe, pas à l'instance. Partagé par tous les objets.
  - **Variable static** : Partagée par toutes les instances de la classe. Accès via `Classe.variable`.
  - **Méthode static** : Appel via `Classe.methode()`. N'a pas accès à `this` ni à `super` (pas d'instance).
  - **Attention Sacha** : Une methde statique comme `main` n'a pas accès aux attributs non statiques.
- `final` (variable) : Constante, assignée 1 seule fois (référence).
  - **Primitif** : Valeur fixe.
  - **Objet** : Référence fixe, mais contenu modifiable (sauf pour les objet immuable (`String`)).
- `final` (méthode) : Ne peut pas être redéfinie (`@Override` interdit).
- `final` (classe) : Ne peut pas avoir de sous-classes.

2.4 Classes Imbriquées (Nested Classes)

Une classe définie à l'intérieur d'une autre classe. Il en existe 4 types.

2.4.1 Classe Interne (Inner Class)

- Définie sans le mot-clé `static`.
- Liée à une instance de la classe externe. Ne peut exister sans elle.
- A accès à **tous** les membres de la classe externe, y compris `private`.
- Syntaxe pour l'instancier depuis l'extérieur : `Outer.Inner inner = outerInstance.new Inner();`

```
class Outer {
    private int x = 10;
    class Inner {
        void display() {
            System.out.println("x = " + x); // Accès direct
        }
    }
}
```

FIGURE 1 – Illustration d'une classe interne

2.4.2 Classe Statique Imbriquée (Static Nested Class)

- Déclarée avec le mot-clé `static`.
- Pas liée à une instance de la classe externe. C'est une classe de haut niveau "rangée" dans une autre.
- A accès uniquement aux membres **statiques** de la classe externe.
- Syntaxe pour l'instancier : `Outer.Nested nested = new Outer.Nested();`

```
class Outer {
    private static int y = 20;
    static class Nested {
        void display() {
            System.out.println("y = " + y); // Accès au static
        }
    }
}
```

FIGURE 2 – Illustration d'une classe imbriquée statique

2.5 Type énuméré (Enum)

Un type énuméré est une classe spéciale qui représente un groupe de **constantes**. C'est la manière la plus propre et sûre de modéliser un ensemble fixe de valeurs (ex : jours de la semaine, états, couleurs).

- **Type-Safe** : Le compilateur garantit qu'une variable de type `Jour` ne peut contenir que les valeurs définies (LUNDI, MARDI, etc.), évitant les erreurs liées à l'utilisation de `String` ou `int`.
- **Plus qu'un simple nom** : Une énumération est une vraie classe. Elle peut avoir des **attributs**, des **méthodes** et un **constructeur**.
- **Constructeur privé** : Le constructeur d'un enum est implicitement **privé**. On ne peut pas créer de nouvelles instances avec `new`.
- **Utilisation** : On y accède via `NomEnum.VALEUR` (ex : `Feux.ROUGE`). Idéal pour les instructions `switch`.

Voici un exemple d'énumération avec des attributs et des méthodes :

```
public enum Feu {
    VERT("Vert", false),
    ORANGE("Orange", false),
    ROUGE("Rouge", true);

    private final String nom;
    private final boolean arretRequis;

    // Constructeur est implicitement private
    Feux(String nom, boolean arretRequis) {
        this.nom = nom;
        this.arretRequis = arretRequis;
    }

    public String getNom() {
        return nom;
    }

    public boolean doitArreter() {
        return arretRequis;
    }
}

// Utilisation
Feux monFeu = Feux.VERT;
if (monFeu.doitArreter()) {
    // ...
}
```

FIGURE 3 – Illustration de le déclaration et de l'utilisation d'un type énuméré

3 Héritage et Polymorphisme

3.1 Définitions

— **Surcharge (Overload)** : Même nom, **paramètres différents**. Résolu à la **compilation** (type statique).

— **Redéfinition (Override)** : Même signature, comportement différent dans sous-classe. Résolu à l'**exécution** (type dynamique).

3.2 Résolution des liens

Soit `A a = new B();` (B étend A).

1. **Type Statique** (A) : Détermine ce qu'on a le *droit* d'appeler. Si la méthode n'existe pas dans A → Erreur compil.
2. **Type Dynamique** (B) : Détermine quelle version est exécutée. Si B redéfinit la méthode, c'est celle de B qui tourne.
3. **Attributs** : Toujours liés au type **statique** (Pas de polymorphisme sur les attributs !).
4. Les méthodes `static` sont des liens résolus statiquement.

```java
class Animal {
    public void manger() {
        System.out.println("L'animal mange.");
    }
}

class Chien extends Animal {
    @Override
    public void manger() {
        System.out.println("Le chien mange sa gamelle.");
    }

    public void aboyer() {
        System.out.println("Woof !");
    }
}

public class TestPolymorphisme {
    public static void main(String[] args) {
        // Type statique: Animal, Type dynamique: Chien
        Animal monAnimal = new Chien();

        // OK: manger() existe dans Animal.
        // La version de Chien est appelée.
        monAnimal.manger();

        // ERREUR DE COMPILATION: aboyer() n'existe pas dans Animal.
        // monAnimal.aboyer();

        // Pour y accéder, il faut une référence de type Chien.
        Chien monChien = (Chien) monAnimal;
        monChien.aboyer(); // OK après cast.
    }
}
```

FIGURE 4 – Exemple développé pour le point 1 de la résolution des liens

3.3 Classes Abstraites vs Interfaces

Classe Abstraite	Interface
`extends` (1 seule)	`implements` (N)
Peut avoir état (attributs)	Pas d'attributs (sauf constantes static final)
Constructeurs possibles	Pas de constructeurs
Méthodes privées/protected OK	Méthodes public par défaut
Relation "EST-UN"	Relation "SE COMPORTE COMME"

TABLE 3 – Différences entre classe abstraite et interface.

Java 8+ : Les interfaces peuvent avoir des méthodes `default` (avec corps) et `static`.
Attention aussi : Une méthode `default` dans une interface **ne peut pas** être déclarée `final`. Une méthode `default` est conçue pour être redéfinie, ce qui est incompatible avec le modificateur `final`.
Attention : Les variables dans une interface sont **toujours** `public static final` (des constantes), on ne peut pas avoir de variable d'instance ou avec la visibilité `default`.

4 Exceptions

4.1 Hiérarchie des Exceptions et Règle Fondamentale

La règle de base du compilateur est la suivante : toute classe qui hérite de `Throwable` est considérée comme **checked**, sauf :

— La classe `Error` et ses descendantes (erreurs système graves).

— La classe `RuntimeException` et ses descendantes (erreurs de programmation).

C'est pourquoi `throw new Exception()` est une opération *checked* : la classe `Exception` n'est ni une `Error`, ni une `RuntimeException`. Le schéma ci-dessous clarifie cette hiérarchie et la distinction.

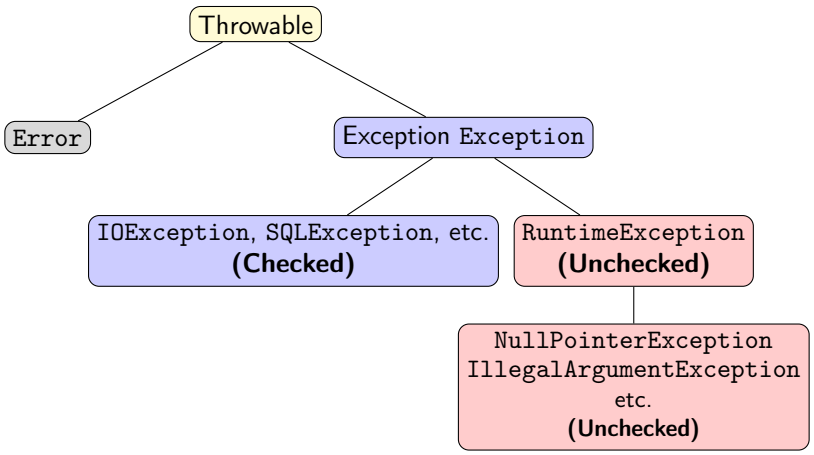

FIGURE 5 – Illustration de hiérarchie de classe

— **Throwable** : Classe racine de tout ce qui peut être "lancé".

— **Error** : Erreurs graves (ex : `OutOfMemoryError`), non destinées à être attrapées.

— **Exception** : Erreurs applicatives.

  — **Branche 'RuntimeException' (Unchecked)** : Erreurs de logique qui ne demandent pas de `try-catch` obligatoire (ex : `NullPointerException`). Le compilateur les ignore.

  — **Les autres branches (Checked)** : Erreurs prévisibles que le compilateur **force** à gérer (ex : `IOException`).

4.2 Rôle du Compilateur

Le compilateur Java applique une règle stricte pour les **checked exceptions** : la règle du "Catch or Specify" (Attraper ou Déclarer).

Si un morceau de code peut lever une *checked exception*, le compilateur **refusera de compiler** tant que vous n'aurez pas fait l'une des deux choses suivantes :

1. **Attraper (Catch)** : Encadrer le code à risque dans un bloc `try-catch`. C'est vous qui gérez le problème.
2. **Déclarer (Specify)** : Ajouter `throws NomDeLException` à la signature de la méthode. Vous déléguez ainsi la responsabilité de la gestion à la méthode appelante.

Question clé : Le compilateur râle-t-il si on utilise `throw` sans `throws` ?

— **OUI**, si l'exception est *checked*. Le code ci-dessous ne compile pas car `Exception` est checked et non déclarée.

```
// ERREUR DE COMPILATION !
void methode() {
    throw new Exception("Ceci est une checked exception");
}
```

Pour corriger : `void methode()throws Exception { ... }`

— **NON**, si l'exception est *unchecked* (`RuntimeException` ou une de ses filles). Le code ci-dessous compile parfaitement.

```
// OK, COMPILE !
void methode() {
    throw new RuntimeException("Ceci est une unchecked exception");
}
```

4.3 Blocs de Gestion

— **try** : Contient le code qui peut lever une exception.

— **catch** : Attrape et gère une exception. On peut avoir plusieurs blocs `catch` pour différents types d'exceptions. **Important :** Toujours attraper les exceptions les plus spécifiques avant les plus générales.

— **finally** : Bloc de code **exécuté dans tous les cas**, qu'une exception soit levée ou non, et même si un `return` est présent dans le bloc `try`. Très utilisé pour libérer des ressources (ex : fermer un fichier).

— **throw** : Pour lancer manuellement une exception (`throw new MyException();`).

— **throws** : Dans la signature d'une méthode pour déclarer les *checked exceptions* qu'elle peut lancer.

```
void readFile() throws IOException {
    FileReader reader = null;
    try {
        reader = new FileReader("file.txt");
        // ... code de lecture
    } catch (FileNotFoundException e) {
        // Gérer le cas où le fichier n'existe pas
        System.out.println("Fichier non trouvé !");
    } finally {
        if (reader != null) {
            reader.close(); // Libération des ressources
        }
    }
}
```

FIGURE 6 – Exemple de bloc de gestion des examens

5 Modèles de Conception (Exam)

5.1 Checklist Exercice Conception

Si on demande de "modéliser" (ex : Termites) :

1. **Identifier les classes** : Noms communs (Termite, Brindille).

2. **Héritage** : "Est un type de..." → `extends`.
3. **Interface** : "Peut être..." ou "Se comporte comme..." → `implements`.
4. **Encapsulation** : Attributs `private`. Accès via Getters/Setters **si nécessaire**.
5. **Collections** : "Contient plusieurs..." → `ArrayList<Type>`.

5.2 Copie Profonde

Pour copier un objet qui contient des objets (références) :

```
public class MaClasse1{
    private MaClasse2 objet;
    private int primitif;

    public MaClasse(MaClasse1 autre) {
        this.primitif = autre.primitif;
        // COPIE PROFONDE:
        this.objet = new MaClasse2(autre.objet); //Il faut que MaClasse2 ait un
            constructeur de copie

        //Remarque : on peut faire autre.objet même si l'attribut est private car on
            est dans la même classe et Java autorise l'accès aux attributs privés
            entre instances de la même classe!
    }
}
```

Ne jamais faire `this.objet = autre.objet` (copie de surface, danger !).

6 Astuces du Corrigé 18-19

— **Passage de paramètres (Toujours par valeur)** : Le passage de paramètres à une méthode en Java se fait toujours "par valeur". Cependant, le comportement diffère entre les types primitifs et les objets.

— **Types primitifs** (`int`, `double`, etc.) : La *valeur* de la variable est copiée. Toute modification de la copie à l'intérieur de la méthode n'affecte pas la variable originale.

```
void method(int i) {
    i = 10; // Ne change pas la variable originale à l'extérieur
}
```

— **Objets** (`String`, `A`, etc.) : La *valeur de la référence* (l'adresse de l'objet) est copiée. Les deux références (originale et copie) pointent vers le **même objet**.

— Si on **modifie l'état** de l'objet via cette référence copiée (ex : `a.setA(5)`), la modification est visible à l'extérieur.

— Si on **réassigne la référence** dans la méthode (ex : `a = new A(10)`), seule la copie de la référence est changée pour pointer vers un nouvel objet. La variable originale à l'extérieur continue de pointer vers l'objet initial.

```
void m(A a_param, int i_param) {
    ++i_param; // Ne change pas l'original
    a_param.setA(a_param.getA() + 1); // CHANGE l'objet original
    a_param = new A(10); // NE CHANGE PAS la variable originale
}
```

— **Math** : `1/2 = 0` (entiers), `1.0/2 = 0.5`.

— **Immutable** : String est immuable. `s.toUpperCase()` ne change pas s, il renvoie un nouveau String.

— **ArrayList** : `List<Truc> l = new ArrayList<>();` (Coder vers l'interface List, pas l'implémentation ArrayList).

— Pour déclarer une variable : `int i = 0;`

— Pour appeler une méthode : `obj.methode(); i++`