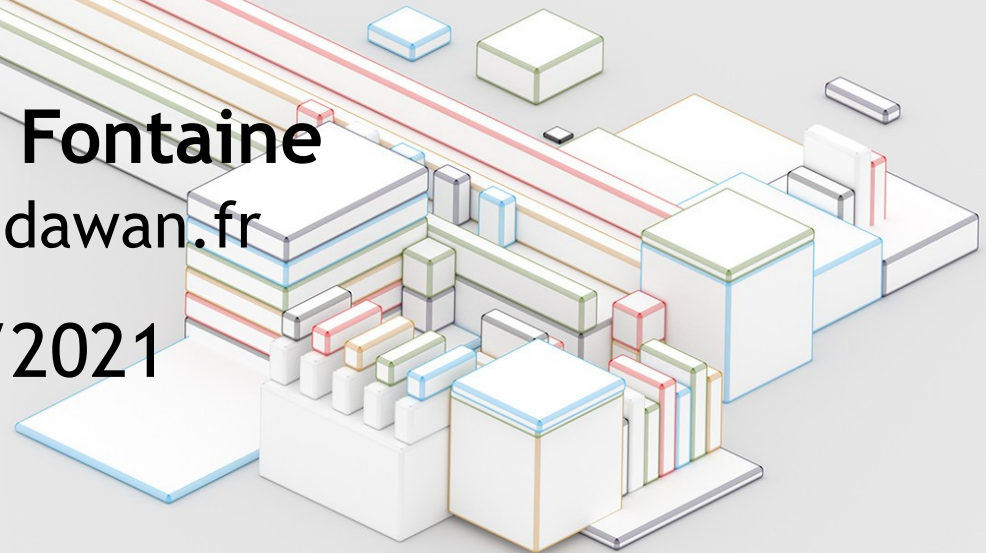


# Préparation de la certification Java SE 8 Programmer1

**Christophe Fontaine**

[cfontaine@dawan.fr](mailto:cfontaine@dawan.fr)

24/12/2021



# Objectifs

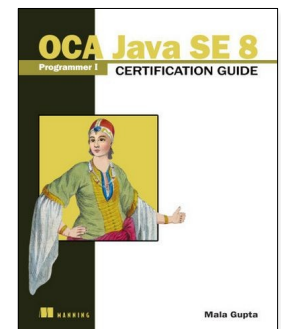
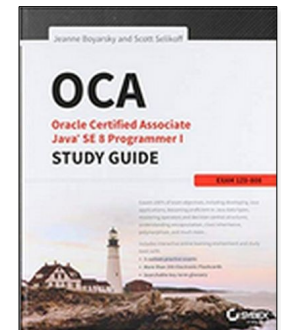
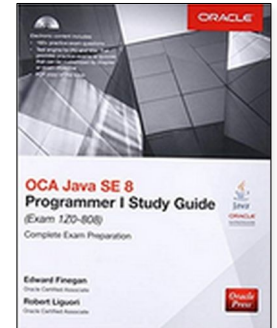
---



- Découvrir le cursus de certification Oracle Java
- Préparer la certification Java SE 8 Programmer 1
  - Révision des thèmes abordés dans l'examen
  - Comprendre le type et le format des questions
  - S'entraîner à répondre à des questions

# Bibliographie

- **OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)**  
Edward G. Finegan et Robert Liguori  
Oracle Press - Septembre 2015
- **OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide (Exam 1Z0-808)**  
Jeanne Boyarsky et Scott Selikoff  
Sybex - Janvier 2015
- **OCA Java SE 8 Programmer I Certification Guide**  
Mala Gupta  
Manning - Septembre 2016



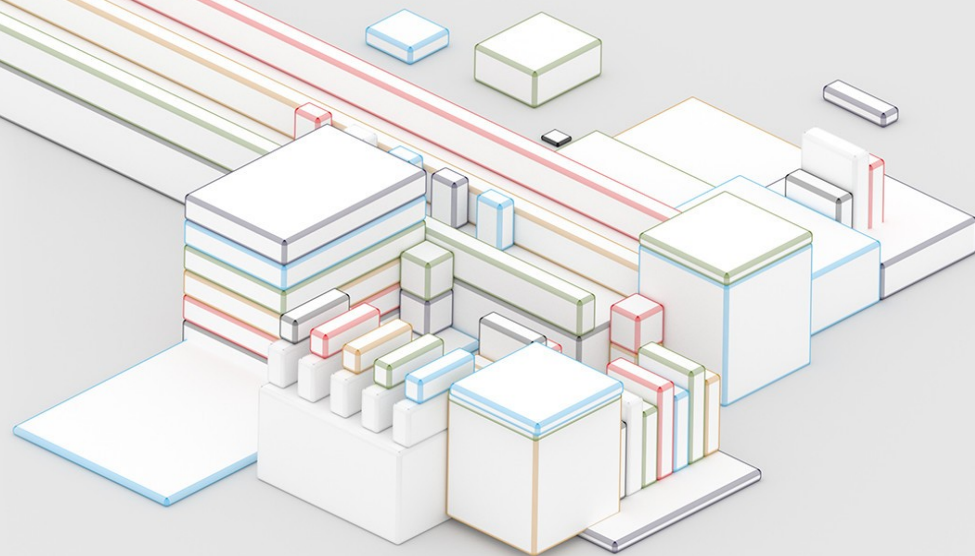
# Plan

---



- Présentation
- Notions de base et Types de données
- Opérateurs et Instructions
- Utilisation de classes de l'API Java
- Méthodes et Encapsulation
- Héritage et Abstraction
- Gestion des exceptions

# Présentation



# Liste des certifications Oracle Java



Certification	N° examen	Examen
Java Foundations Certified, Junior Associate	1Z0-811	Java Foundations
<b>Oracle Certified Associate, Java SE 8 Programmer</b>	<b>1Z0-808</b>	<b>Java SE 8 Programmer I</b>
Oracle Certified Professional, Java SE 8 Programmer	1Z0-809	Java SE 8 Programmer II
Oracle Certified Professional, Java SE 11 Developer	1Z0-819	Java SE 11 Developer
Oracle Certified Professional, Java SE 17 Developer	1Z0-829	Java SE 17 Developer
Oracle Certified Professional, Java EE 7 Application Developer	1Z0-900	Java EE 7 Application Developer

# Java SE 8 Programmer I (1Z0-808)



- **Format** : Questionnaire à choix multiple
- **Durée** : 2h 00
- **Nombre de questions** : 56
- **Note d'obtention** : 65 %
- **Prix** : 228 €
- **Examen en anglais**



[https://education.oracle.com/java-se-8-programmer-i/pexam\\_1Z0-808](https://education.oracle.com/java-se-8-programmer-i/pexam_1Z0-808)



# Exemple de question



Which of the following are valid Java identifiers?  
(Choose all that apply)

- A) \$D
- B) \_helloWorld
- C) false
- D) java.lang
- E) Public
- F) 2000\_s



# Exemple de question

**Which of the following are valid Java identifiers?**

(Choose all that apply)

- A) \$D
- B) \_helloWorld
- C) false
- D) java.lang
- E) Public
- F) 2000\_s

**Réponse : A, B, E**

- **A** et **B** → On peut utiliser `_` et `$` dans les identificateurs
- **E** → Java est sensible à la casse **Public** n'est pas un mot réservé contrairement à **public**

# Exemple de question

**What is the output of the following program?**

```
1: public class WaterBottle {  
2: private String brand;  
3: private boolean empty;  
4: public static void main(String[] args) {  
5: WaterBottle wb = new WaterBottle();  
6: System.out.print("Empty = " + wb.empty);  
7: System.out.print(", Brand = " + wb.brand);  
8: } }
```

- A) Line 6 generates a compiler error
- B) Line 7 generates a compiler error
- C) There is no output
- D) Empty = false, Brand = null
- E) Empty = false, Brand =
- F) Empty = null, Brand = null

# Exemple de question

**What is the output of the following program?**

```
1: public class WaterBottle {  
2: private String brand;  
3: private boolean empty;  
4: public static void main(String[] args) {  
5: WaterBottle wb = new WaterBottle();  
6: System.out.print("Empty = " + wb.empty);  
7: System.out.print(", Brand = " + wb.brand);  
8: } }
```

- A) Line 6 generates a compiler error
- B) Line 7 generates a compiler error
- C) There is no output
- D) Empty = false, Brand = null
- E) Empty = false, Brand =
- F) Empty = null, Brand = null

**Réponse : D**

# Exemple de question



**Which of the following are true statements?**

(Choose all that apply)

- A) Java allows operator overloading
- B) Java code compiled on Windows can run on Linux
- C) Java has pointers to specific locations in memory
- D) Java is a procedural language
- E) Java is an object-oriented language
- F) Java is a functional programming language

# Exemple de question

**Which of the following are true statements?**

(Choose all that apply)

- A) Java allows operator overloading
- B) Java code compiled on Windows can run on Linux**
- C) Java has pointers to specific locations in memory
- D) Java is a procedural language
- E) Java is an object-oriented language**
- F) Java is a functional programming language

**Réponse : B, E**

- **B** → Java est indépendant de la plateforme
- **E** → Java est un langage orienté objet  
( Certaines parties de la programmation fonctionnelle sont  
présent en charge mais dans des classes)

# Conseil pour la révision



- Lire le livre fournit lors de la formation
- Ne faire le questionnaire du livre que lorsque l'on pense maîtriser la partie auquel il se rapporte (si l'on fait trop souvent les questionnaires, on mémorise les réponses et l'on a une fausse impression de progression)
- Ne pas hésiter à exécuter les exemples du livre dans un IDE ou créer ses propres exemples pour faciliter la compréhension

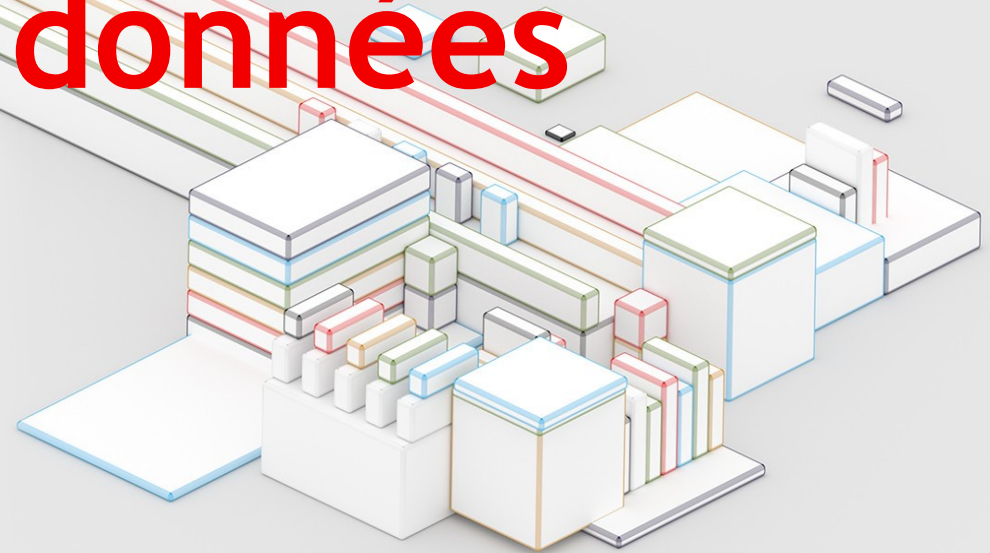
# Conseil pour l'examen



- S' il y a **"est-ce que le code compile"**, parmi les réponses proposées : vérifier que le code compile avant de chercher à répondre à la question → cela permet d'éviter de perdre du temps
- S' il n'y a pas **"Le code ne compile pas"**, parmi les réponses proposées → pas besoin de passer du temps à vérifier la syntaxe
- Si l'on bloque sur une question, on peut la passer et y revenir plus tard
- Répondre à toutes les questions, il n'y a pas de pénalité si la réponse à une question est fausse  
(un peu avant la fin de l'examen, répondre au question qui pose problème même si l'on n'est sûr à 100% de la réponse)
- Gérer le temps en vérifiant périodiquement, le nombre de question et le temps qu'il reste



# Notions de base et Types de données



# Classe et fichier



- Généralement, on a **une classe par fichier .java**
- Si l'on met plusieurs classes dans un fichier, il ne peut contenir qu'**une seule classe public**
- Le nom du fichier doit correspondre au nom de la classe public
- Pour compiler du code java, le fichier doit avoir l'extension **.java**
- La compilation donne un fichier **.class** qui contient du bytecode

# Méthode main

```
public static void main(String[] args) { //... }
```

- La méthode main doit obligatoirement être **public static** et n'a **pas de valeur de retour**
- Les paramètres valident sont:
  - String [] args
  - String args[]
  - String args...
- On exécute un fichier **.class** contenant une méthode main avec la ligne de commande: **java nom\_de\_la\_classe**
- On peut ajouter à cette ligne, des arguments séparés par des espaces, si un argument contient un espace, on l'entoure de "  
**java nom\_de\_la\_classe arg1 arg2 "autre argument"**

# Package et import



- **package** → regroupement logique des classes
- **import** → indique à java dans quel package chercher pour trouver la classe

```
import java.util.Random; //indique où trouver la classe Random
```

- Pour les packages, la règle de nommage est la même que pour les variables, séparé par des points
- \* → wilcard : importe toutes les classes du package

```
import java.util.*
```

**Attention, n'est valable uniquement que pour les classes**

- **java.lang** → package importé automatiquement
- Pour utiliser une classe, on n'a pas besoin de l'importer si l'on est dans le **même package**

# Imports redondant

↳ Suppression des imports inutiles

```
// import redondant java.lang importé automatiquement
import java.lang.System ;
import java.lang.* ;
import java.util.Random ;

// import redondant Random est déjà importé par la ligne
// précédente
import java.util.*

public class ImportExample {
    public static void main(String[] args){
        Random r =new Random() ;
        System.out.println(r.nextInt(10)) ;
    }
}
```

# Conflit sur les noms

- Une des raisons d'utiliser les package est que plusieurs classes peuvent porter le même nom  
ex : `java.util.Date` et `java.sql.Date`
- **Problème** → lorsque l'on veut utiliser plusieurs classes ayant le même nom dans un programme

```
import java.util.*;  
import java.sql.*;           // NE COMPILE PAS:  
                             le type Date est ambiguë  
  
import java.util.Date;  
import java.sql.Date        // NE COMPILE PAS:  
                             import java.sql.Date rentre  
                             en collision avec un autre import
```

# Conflit sur les noms

- Si on utilise l'objet **Date** du package **java.util**

```
import java.util.Date;  
import java.sql.*;           // pour utiliser les autres classes  
                             // du package sql autre que Date
```

- Si l'on doit utiliser **2 classes** avec le **même nom**

```
import java.util.date;  
Date date                // Date du package util  
java.sql.Date sqlDate // défini la classe avec le nom complet
```

ou on utilise pour les deux le **nom complet**

```
java.util.Date date;  
java.sql.Date sqlDate;
```



# Créer un nouveau package



- Le nom du package est lié à la structure des répertoires dans le système de fichier de l'ordinateur
- On se trouve dans **c:\temp**

**c:\temp\packagea\ClasseA.java**

```
package packagea;  
public class ClasseAe {  
}
```

**c:\temp\packageb\ClasseB.java**

```
package packageb;  
import packagea.ClasseA;  
public class ClasseB {  
    public static void main(Strig[] args) {  
        ClasseA a;  
    }  
}
```

- **default package** → package spécial sans nom, à n'utiliser que pour du code temporaire

# Constructeur

- Un objet est une instance d'une classe

```
Random r = new Random();
```

constructeur

déclaration de "type Random" qui contiendra la référence

- Un constructeur a pour but d'initialiser les attributs
  - Il a le même nom que la classe
  - Il n'a pas de type de retour
- **S'il n'y a pas de constructeur défini**, java en crée un par défaut (un constructeur sans paramètre qui ne fait rien)

**Attention**, `public void Random(){ }` → Ce n'est pas un constructeur ( il y a un type de retour), mais une méthode

# Initialisation des attributs

- On peut initialiser directement à la déclaration des attributs

```
public class Chicken() {  
    int numEggs = 0;  
}
```

- **Bloc d'initialisation**

C'est un bloc de code en dehors d'une méthode



```
class Test {  
    int a = 1;           // 1 initialisation du champ → a vaut 1  
    public Test(){  
        a = 3 ;         // 3 appel du constructeur → a vaut 3  
    }  
    {  
        a=2;            // 2 bloc d'initialisation → a vaut 2  
    }  
    public static void main(String arg[]){  
        Test t=new Test() ; } }
```

# Ordre d'initialisation



- 1) Toutes les **initialisations des variable d'instances** et les **blocs d'initialisation** sont exécutées dans l'ordre où ils apparaissent dans le code
- 2) Les **constructeurs** sont appelés ensuite

**Attention**, à l'ordre d'exécution, on ne peut pas faire référence à une variable qui n'a pas été initialisée

```
{  
    System.out.println(name); // ne compile pas  
}  
private String name = "Donald" ;
```

# Type

- 8 types primitifs:

Type	Taille	Valeur
boolean		true ou false
byte	entier signé sur 8 bits	-128 à 127
short	entier signé sur 16 bits	
int	entier signé sur 32 bits	
long	entier signé sur 64 bits	
float	valeur réel sur 32 bits	
double	valeur réel sur 64 bits	
char	caractère unicode sur 16 bits	

- Littéral long → **L**, float → **f**

```
long l = 123456789000L;  
float f = 20.0f;
```

- **Changement de base**

- **0**17 → octal
- **0x**ff → hexadécimal
- **0b**10011 → binaire

- **\_ pour améliorer la lecture des littéraux**

```
int million =1_000_000;
```

On peut ajouter des \_ partout sauf :  
au début, à la fin, avant et après la virgule

\_1000    1000\_    100\_.00    100.\_0;    → FAUX

- **Type référence**

Il fait référence à un objet

Un objet ne peut être accessible que par une référence

Si elle ne “référence” rien, elle a pour valeur **null**

# Déclarer et initialiser des variables



- **Déclarer une variable** → définit le type de la variable et lui donner un nom

```
int i;
```

- **Initialiser une variable** → donner une valeur à la variable

```
i = 0;
```

- **On peut initialiser une variable à la déclaration**

```
String str="Mike";
```

- **Déclaration de variables multiples**

Elles doivent être de même type et séparées par ,

```
String x1, x2;  
String x3 = "aze", x4 = "qs";  
int i1 = 0, i2, i3 = 8;           // attention i2 n'est pas  
                                // initialisé
```



# Règles de nommages



- Le nom doit commencer par : une **lettre**, **\_** ou **\$**
- Les nombres sont autorisés sauf en tête
- Le nom ne doit pas être un **mot réservé**

**Attention** : même si ils ne sont pas utilisés, **const** et **goto** sont des mots réservés

- Valable pour les :
  - attributs
  - méthodes
  - variables
  - classes

# Initialisation par défaut des variables

- **Variable locale**

- Elles sont définies à l'intérieur d'une méthode
- Elles doivent obligatoirement être initialisées avant d'être utilisées
- Elles n'ont pas de valeur par défaut

Les arguments d'une méthode sont des variables locales

- **Attribut d'instance et de classe**

Les attributs d'instance et de classe ont une valeur par défaut

Type	Valeur par défaut
boolean	false
byte, short, int, long	0
float, double	0.0
char	\u0000
référence	null

# Porté des variables

- Chaque bloc de code { } à sa propre portée
- Quand des blocs contiennent d'autre bloc :  
Les blocs contenus peuvent faire référence aux variables du bloc conteneur mais pas l'inverse

```
{  
    int a = 10;  
    if (a < 24) {  
        int b = a + 4;  
    }  
    System.out.println(b); // ne compile pas  
}
```

**variable locale** → de sa déclaration à la fin du bloc

**variable d'instance** → de sa déclaration jusqu'à la  
destruction de l'objet par le garbage collector

**variable de classe** → de sa déclaration jusqu'à la fin du  
programme

# Ordre des éléments dans une classe

Élément	Obligatoire	Emplacement
<b>package</b>	non	1ère ligne du fichier
<b>import</b>	non	juste après le <b>package</b>
<b>class</b>	<b>oui</b>	juste après les <b>imports</b>
<b>attributs</b>	non	partout dans la classe
<b>méthodes</b>	non	partout dans la classe

```
import java.util.* ;  
package structure ;      // Ne compile pas  
String name ;           // Ne compile pas  
public class Meerkat{  
  
}
```

# Destruction des objets



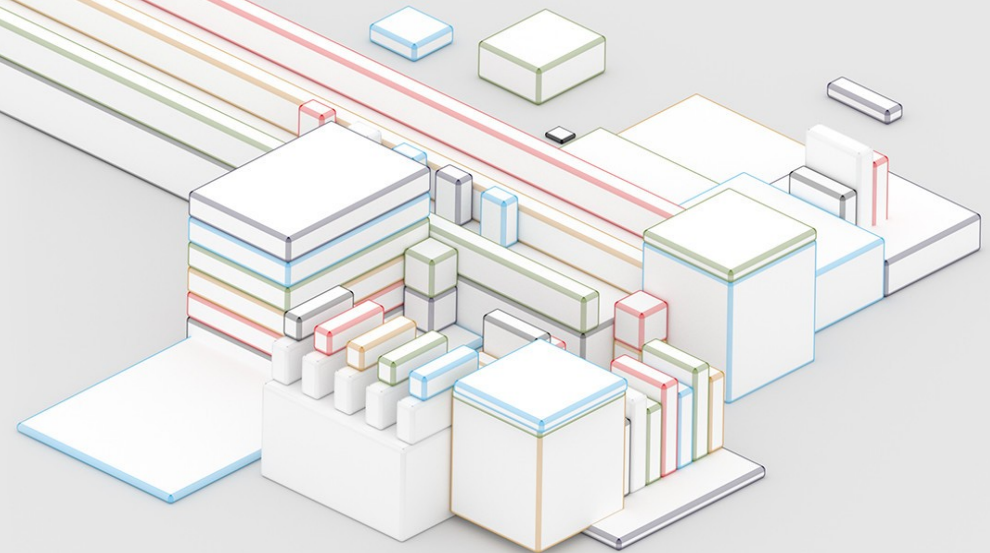
- Tous les objets sont stockés dans le tas (heap)
- **garbage collector** → détruit automatiquement les objets devenus inutiles
- Un objet est éligible à la destruction par le garbage collector lorsque:
  - L'objet n'a plus de référence qui pointe sur lui
  - Toutes les références sont hors de portées
- **System.gc()** → appel du garbage collector (n'est pas garantie d'être exécuté, peut-être ignoré par java)
- **Méthode finalize** → Appelée si le garbage collector essaye de collecter l'objet. Si la collecte échoue, la méthode ne sera pas appelée une seconde fois, lors de la prochaine collecte  
↳ **finalize** peut être appelé entre **0** et **1** fois

# Bénéfice de java



- **Orienté objet**
- **Encapsulation**
- **Indépendant de la plateforme:** compilé un fois, peut être exécuté sur tous les systèmes disposant d'une JVM
  - ↳ Write once, run everywhere
- **Robuste:** comparé au C++, pas de fuite de mémoire
  - ↳ garbage collector
- **Simple:** plus simple que le C++, pas de pointeur, pas de surcharge d'opérateur (operator overloading)
- **Sécurisé:** le code est exécuté dans la JVM (sandbox)

# Opérateurs et Instructions





# Opérateurs



- **Java a 3 types d'opérateurs**

unaire → 1 opérande

binaire → 2 opérandes

ternaire → 3 opérandes

- **Opérateur arithmétique (opérateur binaire)**

- + addition ou concaténation avec les chaînes de caractère

- soustraction

- \* multiplication

- / division

- % modulo, reste de division entière, fonctionne aussi avec les nombres négatifs et les entiers en virgule flottante

# Promotion numérique

## 1) Si 2 valeurs ont 2 types différents

↳ java va promouvoir une des valeurs vers le plus grand des 2 types

```
float f = 1.0f;  
double d = 2.0;  
double k = f + d ;    // f est promu en double
```

## 2) Si une valeur est entière et l'autre à virgule flottante

↳ la valeur entière est promue en virgule flottante

```
int i=10;  
double d=320;  
double r=i+d;    // i est promu en double
```

# Promotion numérique

- 3) **byte, short, char** sont promues en **int** à chaque fois qu'ils sont utilisés avec un opérateur arithmétique binaire

```
short s1 = 4;  
short s2 = 10;  
short s = s1 + s2;           // FAUX  
int i = s1 + s2;              // s1 et s2 sont promu en int
```

- 4) Après toutes les promotions numériques, le **résultat aura le même type que les opérandes promus**

# Opérateur unaire

- + nombre positif (sans, le nombre est considéré comme positif)
- nombre négatif
- ! inversion d'une valeur booléenne
- ++ incrémentation
- décrémentation

- **Pré-incrémentation**

```
int x = 2;  
int res = ++x;  
// ...
```

équivalent à  $\left\{ \begin{array}{l} x=x+1; \text{ // } x \text{ vaut } 3 \\ res=x; \text{ // } res \text{ vaut } 3 \end{array} \right.$

- **Post-incrémentation**

```
int x = 2;  
int res = x++;  
// ...
```

équivalent à  $\left\{ \begin{array}{l} res=x; \text{ // } res \text{ vaut } 2 \\ x=x+1; \text{ // } x \text{ vaut } 3 \end{array} \right.$

# Casting explicite

- type numérique: **le plus grand vers le plus petit**

```
int j = 12;  
short s = (short) j;  
  
double d = 50.0 ;  
float f= (float) d ;
```

- **réel vers un entier**

```
double d = 10.0;  
int i = (int) d;
```

# Opérateur d'affectation

```
int x = 0 ;  
int y = 1.0 ;           // Ne compile pas  
short z = 1921222;      // Ne compile pas  
int i = 9f ;           // Ne compile pas  
long l = 192301398193810323 // Ne compile pas
```

- **Opérateur d'affectation composé**

`+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`

`a += 10` équivaut à `a = a + 10;`

**Attention**, les opérateurs d'affectation composés réalisent une conversion automatique

```
long x = 10;  
int y = 5;  
y = y * x;      // Erreur  
y *= x;         // OK, pas d'erreur
```

# Opérateur relationnel



- < inférieur
- <= inférieur égal
- > supérieur
- >= supérieur égal
- Compare 2 expressions et retourne un booléen
- **a instanceof b** → vrai, si a est une instance d'une **classe**, **sous-classe** ou d'une classe qui implémente un *interface* nommé avec b

# Opérateur d'égalité

**==**      égalité

**!=**      différence

## 3 scénarios d'utilisation de l'opérateur égalité

- comparer 2 types primitifs

**Attention**, promotion numérique `5==5.0` → vraie

- comparer 2 valeurs boolean
- comparer 2 objets (incluant null et les valeurs String)  
pour les objets l'égalité se fait sur les références

```
File x= new File("MyFile.txt");  
File y= new File("MyFile.txt");  
File z=x;  
// x==y      → faux  
// x==z      → vrai
```



# Opérateur logique

&	et
	ou
^	ou exclusif

## Opérateur "court circuit"

- **&&** → faux, dès qu'une valeur est fausse et ne poursuit pas le reste de l'évaluation de l'expression

```
if(x! = null && x.getvalue()<5)  
// x.getvalue()<5 n'est pas évaluer  
// évite une exception nullpointer
```

- **||** → vraie, dès qu'une valeur est vraie et ne poursuit pas le reste de l'évaluation de l'expression

# Instruction conditionnelle



- **if**

```
if (expression booléenne) {  
    // si l'expression est vraie  
}
```

```
if (expression booléenne) {  
    // si l'expression est vraie  
} else {  
    // si l'expression est  
    // fausse  
}
```

```
if(expression booléenne 1){  
    // si l'expression1 est vraie  
} else if(expression booléenne 2) {  
    // si l'expression2 est vraie  
} else {  
    // si les expressions 1 et 2  
    // sont fausses  
}
```

**Attention**, vérifier que le test est une expression boolean

- **Opérateur ternaire**

expressionBooleenne ? si expression vraie : si expression fausse;

# Instruction conditionnelle : switch



```
switch (variableAtester) {  
  case expressionConstante1:  
    // ...  
  break;  
  case expressionConstante2:  
    // ...  
  break;  
  default:  
    // ...  
}
```

- **break** et **default** sont optionnels
- la variable testée doit être de type:
  - byte, short, char, int
  - les wrapper associés (Byte, Short, Character, Integer)
  - String
  - énumération

# Instruction conditionnelle : switch



- pour les **case**, la valeur doit être constante au moment de la compilation : **littéral**, **énumération**, **final** (constante)
- **default** n'est pas obligatoirement placé en fin de switch

**Attention**, à la présence de **break** et à l'ordre des instructions

```
switch(i) {  
  case 0:  
    System.out.print("A");  
  default:  
    System.out.print("B");  
  case 6:  
    System.out.print("C");  
}  
  
// pour i=4 ou i=5  → BC  
// pour i=0         → ABC  
// pour i=6         → C
```

# Boucle: while et do while



- **while**

```
while(expression booléenne){  
    // instructions à exécuter  
}
```

Tant que la condition est vérifiée, le bloc d'instructions est exécuté

- **do while**

```
do {  
    // instructions à exécuter  
} while (expressionBooléenne);
```

Identique à **while**, sauf que le test est réalisé après l'exécution du bloc → **au moins exécuté une fois**

# Boucle: for

```
for(initialisation ; condition ; opération){  
    // instructions à exécuter  
}
```

1) **initialisation** est exécutée

2) si la **condition** est fausse → on sort de la boucle

3) le bloc d'instruction est exécuté

4) **opération** est exécutée

On peut ajouter plusieurs termes à la déclaration d'initialisation et d'opération

Ils doivent être séparés par des virgules et pour l'initialisation et ils doivent être du même type

```
int x = 0 ;  
for (long y = 0, z = 4; x < 5 && y < 10; x++, y++) {  
    // ...  
}
```

# Boucle : foreach

```
for (int variable : tableau) {  
    // instructions à exécuter  
}
```

- **Itération complète (foreach)**

Pour l'itération d'un tableau et d'une collection d'objet  
(qui implémente `java.lang.iterable`)

```
String[] names=new String[3];  
names[0]="pascal";  
names[1]="alain";  
names[2]="nelly";  
for(String name:names){  
    System.out.println(name + ", ");  
}  
// affiche : pascal, alain, nelly,
```

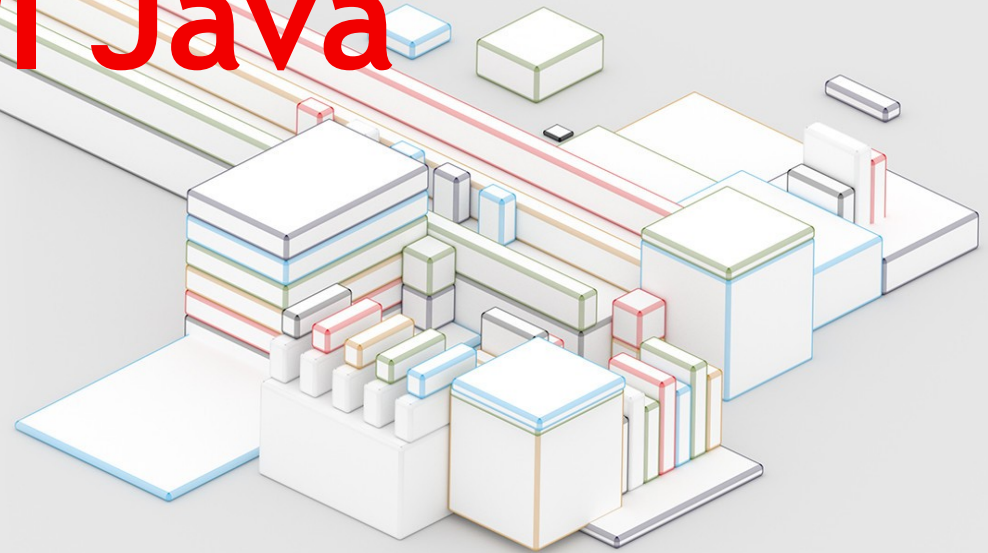
# Instructions de saut



- On peut définir des labels qui seront utilisés avec break et continue → NOMLABEL:
- **break**
  - n'est pas utilisable avec if(uniquement switch et boucle)
  - **break;** → sort de la boucle la plus proche
  - **break label;** → idem, en se branchant sur le label
- **continue**
  - uniquement utilisable dans une boucle
  - **continue;** → finir une itération de la boucle
  - **continue label;** → idem, en se branchant sur le label



# Utilisation de classes de l'API Java



# Concaténation

## L'opérateur +

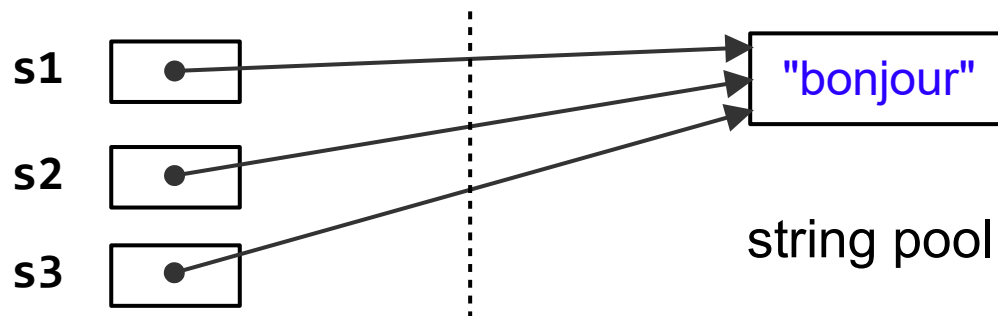
- 1) Si les opérandes sont numériques → **Addition**
- 2) Si l'un des opérandes est une chaîne → **Concaténation**
- 3) L'expression est évaluée de gauche à droite

```
String str="A"+"b"+3;           // donne Ab3  
String str2= 1 + 2 + "c";       // donne 3c
```

# Les chaînes de caractère

- Les chaînes de caractère sont **immuables**  
↳ une fois créé, elles ne **sont plus modifiables**
- String pool → contient toutes **les chaînes de caractère littéral** qui apparaissent dans le programme

Chaque chaîne littéral y est stockée **de façon unique**  
(3 chaînes de caractère littérales "bonjour" dans le programme → un seul objet dans le string pool)



```
String name = "personne"; // stocké dans le stringpool  
String name = new String("Test"); // stocké dans le (heap)
```

# Principale méthode de la classe String



- L'index d'une chaîne commence à 0
- **length()** → retourne la longueur de la chaîne

```
int l = "marcel".length(); // 6
```

- **indexOf(int ch)**  
**indexOf(String str)**  
↳ retourne l'index du 1<sup>er</sup> caractère ou de la 1<sup>ère</sup> chaîne de caractère correspondant

```
int ic = "marcel".indexOf('c'); // 3  
int is = "1232323".indexOf("23"); // 1
```

**indexOf(char ch, int fromIndex)**  
**indexOf(String str, int fromIndex)**

↳ idem mais en commençant à partir de l'indice fromIndex

**Si rien ne correspond → retourne -1**

# Principale méthode de la classe String



- **char charAt(int index)**  
↳ retourne le caractère correspondant à l'index  
si `index > length()` ↳ lance une exception **indexOutOfBoundsException**

```
char c = "marcel".charAt(2);           // r
```

- **String substring(int beginIndex)** → extrait une sous-chaîne qui commence à partir de `beginIndex`

```
String str = "12345678".substring(6); // 78
```

**String substring(int beginIndex, int endIndex)**

↳ extrait une sous-chaîne qui commence à partir de `beginIndex` et finit à partir de `endIndex` (**exclut**), peut-être égal à `length()`

# Principale méthode de la classe String



```
String str="12345678".substring(1,3);    // 23  
String str2="12345678".substring(5,8);    // 678
```

si beginIndex == endIndex → retourne une chaîne vide  
si endIndex > length → lance une exception

- **String toLowerCase()**  
↳ retourne la chaîne en minuscule
- **String toUpperCase()**  
↳ retourne la chaîne en majuscule
- **boolean equals(Object obj)**  
↳ teste l'égalité de 2 chaînes
- **boolean equalsIgnoreCase(String str)**  
↳ teste l'égalité de 2 chaînes en ignorant la casse

# Principale méthode de la classe String



- `boolean startsWith(String prefix)`  
↳ teste si la chaîne commence par le prefix
- `boolean endsWith(String suffix)`  
↳ teste si la chaîne finie par le suffix
- `boolean contains(String str)`  
↳ teste si la chaîne contient la sous-chaîne
- `String replace(char oldCh, char newChar)`  
`String replace(CharSequence oldChar, CharSequence newChar)`  
↳ remplace le caractère ou `CharSequence` `oldChar` par `newChar` (`CharSequence` → `String` ou `StringBuilder`)
- `String trim()` → retire les caractères blancs (espace, `\r`, `\n`, `\t`) en début et en fin de chaîne

# Principale méthode de la classe String



- On peut chaîner les méthodes

```
String result = "Animal".trim().toLowerCase()  
.replace('a', 'A'); // donne Animal
```



# StringBuilder

- Un StringBuilder n'est pas immuable
  - la chaîne peut-être modifiée
  - pas de création de chaîne intermédiaire
- **Création de StringBuilder**

```
// Créer un StringBuilder vide
StringBuilder sb1 = new StringBuilder();
// Créer un StringBuilder initialisé avec une chaîne
StringBuilder sb2 = new StringBuilder("Animal");
// Créer un StringBuilder en réservant 10 emplacements
StringBuilder sb3 = new StringBuilder(10);
```

- **Principale méthode de StringBuilder**  
**charAt(), indexOf(), lenght() et substring()**  
↳ identique à String

# StringBuilder



- `StringBuilder append(String str)`  
↳ ajoute la chaîne à la fin de `StringBuilder`  
Il y a plus de 10 méthodes `append` avec différents paramètres
- `StringBuilder insert(int offset, String str)`  
↳ insère la chaîne à l'index `offset`, comme `append`, la méthode `insert` est surchargée
- `StringBuilder deleteCharAt(int index)`  
↳ permet de supprimer un caractère à l'indice `index`
- `StringBuilder delete(int start, int end)`  
↳ supprime les caractères des indices `start` à `end` (exclut)
- `StringBuilder reverse()`  
↳ inverse l'ordre des caractères
- `toString()` → convertit un `StringBuider` en `String`

# Égalité de 2 chaînes

```
String x = "Azerty";  
String y = "Azerty";  
String z = new String("Azerty");  
String k = "Azerty".trim();
```

- **x==y** vrai, car les chaînes sont littérales x et y font référence à la même chaîne "Azerty" dans le string pool
- **x==z** faux, comparaison de référence différente (string pool et heap)
- **x==k** faux, comparaison de référence différente "Azerty".trim(); crée une nouvelle chaîne
- **x.equals(y)**, **x.equals(z)** et **x.equals(k)** vrai comparaison d'objet, compare le contenu de la chaîne

# Tableau

- **Créer un tableau**

```
int[] num = new int[3];
```

Les éléments du tableau sont initialisés avec la valeur par défaut du type, pour int→0

```
int[] num2 = new int[] { 4, 5, 6, 7 };  
int[] num3 = { 4, 5, 6, 7 };
```

[ ] peut-être placé avant ou après le nom int num [ ] équivaut à int [ ] num

- **Utiliser un tableau**

```
System.out.println(num3[1]);           // affiche 5
```

- **length → taille du tableau**

```
System.out.println(num3.length);       // affiche 4
```

# Classe utilitaire Arrays

- La Classe **Arrays** permet de manipuler des tableaux

```
import java.util.Arrays;
```

- Trier un tableau (dans l'ordre alphabétique)

```
int tab[] = { 5, 1, 6, 9 };  
Arrays.sort(tab);
```

- Recherche dans un tableau trié

```
int binarySearch(tableau, valeur)
```

- Si le tableau n'est pas trié → le résultat est imprévisible
- Si la valeur est trouvée → retourne l'index
- Sinon, retourne une valeur négative, qui a pour valeur l'index où il pourrait être inséré tout en préservant l'ordre du tableau multiplié par -1 en ajoutant -1

# Tableau multidimensionnel

- **Créer un tableau multidimensionnel**

- tableau à 2 dimensions

```
int [][] vars1;  
int vars2 [][];  
int [] vars3 [];  
String [][] rectangle= new String[3][2];
```

- tableau à 3 dimensions

```
int [] space [][];  
int space2 [][][];
```

- tableau asymétrique

```
int [][] tabDiffSize={{1,4,4},{3},{9,4},{8,4,5,6}};  
int [][]args=new int[4][];  
args[0]=new int[5];  
args[1]=new int[3];
```

# Tableau multi dimension

- **Utiliser un tableau multidimensionnel**

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
    for (int j = 0; j < twoD[i].length; j++) {
        System.out.println(twoD[i][j]);
    }
    System.out.println();
}
```

ou

```
int[][] twoD = new int[3][2];
for (int[] inner : twoD) {
    for (int num : inner) {
        System.out.println(num);
    }
    System.out.println();
}
```

# ArrayList

- ArrayList → peut être vu comme un tableau dont la taille peut changer

```
import java.util.ArrayList;
```

- Créer un ArrayList

```
// Crée un ArrayList vide  
ArrayList list= new ArrayList();  
// Crée un ArrayList en réservant 10 emplacements  
ArrayList list2=new ArrayList(10);  
// Crée un ArrayList à partir d'un autre ArrayList  
ArrayList list3=new ArrayList(list2);
```

avec les génériques

```
// Crée une ArrayList de Chaine de caractère vide  
ArrayList<String> list= new ArrayList<String>();  
ArrayList<String> list2=new ArrayList<>();
```



# Principale méthode d'un ArrayList



- `boolean add(E element)`  
↳ ajouter un élément à la fin (retourne toujours true)
- `void add(int index, E element)` → ajouter un élément à la position index
- **Attention**, à l'utilisation des génériques

```
ArrayList list=new ArrayList();  
list.add("hawk");  
// Compile : le tableau stocke des Objects  
list.add((Boolean.true);  
ArrayList<String> list2=new ArrayList();  
list2.add("hawk");  
// Ne Compile pas : on ne peut ajouter que des chaînes  
list2.add((Boolean.true);
```

# Principale méthode d'un ArrayList



- `boolean remove(Object obj)`  
↳ retire l'élément de la liste, retourne true si l'objet est retiré
- `E remove(int index)`  
↳ retire l'élément spécifier par l'index et le retourne
- `E set(int index, E newElement)` → remplace un élément dans l'ArrayList et retourne l'ancien élément
- `boolean isEmpty()` → retourne true si ArrayList est vide
- `int size()` → retourne la taille de l'ArrayList
- `boolean contains(Object object)`  
↳ retourne vrai si l'objet se trouve dans la liste

# Principale méthode d'un ArrayList



- `void clear()` → vide l'ArrayList
- `boolean equals()` → compare 2 listes et retourne true, si elle contient les même éléments dans le même ordre
- **Conversion List → tableau**

```
ArrayList<String> list = new ArrayList<>();  
list.add("aa");  
list.add("bb");  
list.add("cc");  
String[] string2Array = list.toArray(new String[0]);
```

Pour **0** → retourne un nouveau tableau à la taille de la liste

Si la taille du tableau est  $>$  ou  $=$  à la taille de la liste

↳ retourne le tableau, sinon un nouveau tableau est créé

# Principale méthode d'un ArrayList



- **Conversion tableau → List**

```
Integer array[] = { 1, 4, 6 };  
List<Integer> list = Arrays.asList(array);  
// Accepte les paramètres variables  
List<Integer> listPv = Arrays.asList(1, 2, 4, 5);
```

- **Trier un ArrayList**

`Collections.sort( )`

```
ArrayList<String> lst=new ArrayList<>();  
lst.add("C");  
lst.add("A");  
lst.add("B");  
Collections.sort(lst);    // Trie la liste
```

# Les classes Wrapper

- Objets qui correspondent aux types primitifs

boolean	Boolean
byte	Byte
short	Short
int	Integer
float	Float
double	Double
char	Character

- Permet de convertir une chaîne de caractère en primitive **parse** ou en classe wrapper **valueOf**

```
int primitive = Integer.parseInt("123");  
Integer wrapper = Integer.valueOf("123");  
// Lance une exception NumberFormatException  
int erreurConv = Integer.parseInt("a");
```

**Attention, pas de méthode parse et valueOf pour Character**

# Autoboxing

- L'autoboxing, c'est la conversion automatique d'une valeur primitive en une classe wrapper

```
List<Double> weights = new ArrayList<>();  
weights.add(50.0) // création automatique d'un objet Double  
double first = weights.get(0);
```

- **Attention**, l'index à la priorité sur l'autoboxing avec la méthode **remove**

```
List<Integer> numbers = new ArrayList<>();  
numbers.add(1);      // index 0  
numbers.add(2);      // index 1  
numbers.remove(1);   // Supprime l'objet Integer à l'index 1
```

pour forcer l'utilisation du wrapper

↳ `numbers.remove(new Integer(2));`

# Date et heure

```
import java.time.*
```

**LocalDate**                      contient uniquement la date

**LocalTime**                      contient uniquement l'heure

**LocalDateTime**                  contient la date et l'heure

- La méthode statique **now()**  
↳ donne la date et l'heure courante
- La méthode statique **of()**  
↳ permet de créer une date ou une heure spécifique

**Attention**, on ne peut pas créer l'objet **LocalDate**, **LocalTime** et **LocalDateTime** avec **new** uniquement à partir des méthodes static **now** et **of**

- **LocalDate**

Le mois peut être passé soit en entier, soit sous forme d'énumération Month (Month.JANUARY ... Month.DECEMBER)

- `of(int year, int month, int dayOfMonth)`
- `of(int year, Month month, int dayOfMonth)`

- **LocalTime**

- `of(int hour, int minute)`
- `of(int hour, int minute, int second)`
- `of(int hour, int minute, int second, int nanoSecond)`

- **LocalDateTime**

- `of(LocalDate date, LocalTime time)`
- des méthodes combinant des paramètres de LocalTime et LocalDate ...



# Manipuler les dates et l'heure



- La date et l'heure sont immuables (comme String)
- On manipule une date et un temps :
  - avec les méthodes **plus**: `plusYears()`, `plusMonths()`, `plusWeeks()`, `plusDays()`, `plusHours()`, `plusMinutes()`, `plusSeconds()`, `plusNanos()`
  - avec les méthodes **moins**: `minusYears()`, `minusMonths()`, `minusWeeks()`, `minusDays()`, `minusHours()`, `minusMinutes()`, `minusSeconds()`, `minusNanos()`

```
LocalDate date= LocalDate.of(2020, Month.JANUARY, 20);  
// date → 20/04/2020  
date= Date.plusMonths(3) ;  
// La date n'est pas modifiée (immuable)  
date.plusDays(10) ;  
// Ne compile pas (pas de minute dans LocalDate)  
date=date.plusMinutes(1) ;  
// date → 19/03/17  
date= date.minusDays(1).plusMonths(2).minusYears(3) ;
```

# Periods

- Contient une période  
période d'un mois → `Period m = Period.ofMonths(1);`  
`ofYears()`, `ofMonths()`, `ofWeeks()`, `ofDays()`,  
`of(year, month, day)`
- On ne peut pas chaîner les Periods (sinon, seulement la dernière est prise en compte)

```
LocalDate date = LocalDate.of(2015, 1, 20);
Period period = Period.ofMonths(1); // période d'un mois

// affiche 20/02/2015
System.out.println(date.plus(period));

// période d'un an
period = Period.ofWeeks(4).ofMonths(3).ofYears(1);
LocalTime time = LocalTime.of(6, 15);
period = Period.ofMonth(2);

// UnsupportedOperationException
time = time.minus(period);
```

# Formater la Date et l'heure



- La classe **DateTimeFormatter** permet de formater l'heure et la date

```
LocalDate date=LocalDate.of(2020,Month.JANUARY,20);
LocalTime time=LocalTime.of(11,12,34);
LocalDateTime dateTime=LocalDateTime.of(date, time);

// ISO
date.format(DateTimeFormatter.ISO_LOCAL_DATE); // 2020-01-20
time.format(DateTimeFormatter.ISO_LOCAL_TIME); // 11:12:34
dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
// ↳ 2020-01-20T11:12:34

// Format prédéfinie (pour l'examen uniquement SHORT et MEDIUM)
DateTimeFormatter shortDateTime=
DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
shortDateTime.format(dateTime); // 1/20/20
shortDateTime.format(date); // 1/20/20
shortDateTime.format(time);
```

# Formater la Date et l'heure



```
dateTime.format(shortDateTime); // 1/20/20
date.format(shortDateTime); // 1/20/20
time.format(shortDateTime); // UnsupportedOperationException

DateTimeFormatter mediumDateTime =
DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
mediumDateTime.format(dateTime); // Jan 20, 2020 11:12:34 AM
```

- **Méthodes ofLocalized**

DateTimeFormatter f=DateTimeFormatter.          (FormatStyle.SHORT);

	f.format(localDate)	f.format(localDateTime)	f.format(localTime)
<b>ofLocalizedDate</b>	Affiche l'objet en entier	Affiche uniquement la date	<b>Runtime exception</b>
<b>ofLocalizedDateTime</b>	<b>Runtime exception</b>	Affiche l'objet en entier	<b>Runtime exception</b>
<b>ofLocalizedTime</b>	<b>Runtime exception</b>	Affiche uniquement l'heure	Affiche l'objet en entier

# Formater la Date et l'heure



- **Format personnalisé**

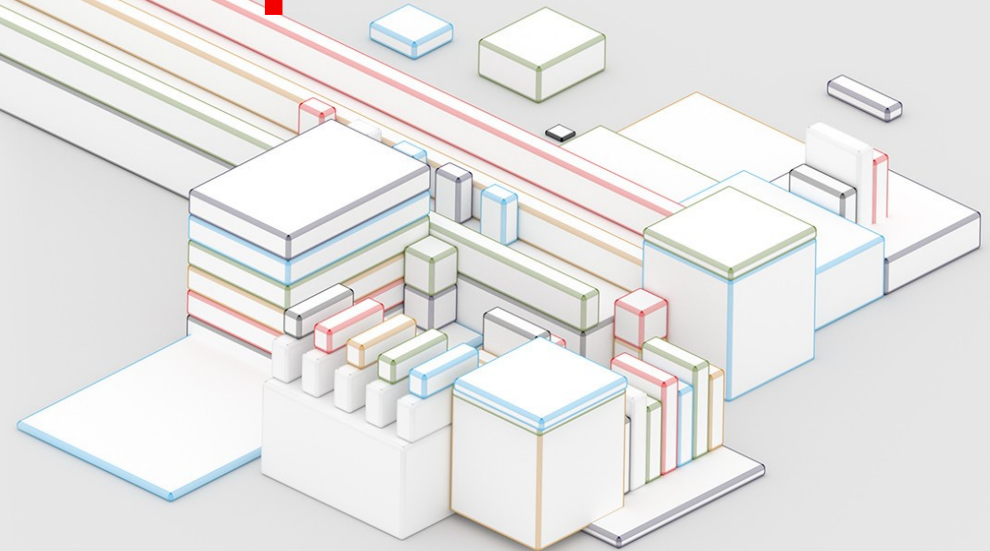
```
DateTimeFormatter customFormat=DateTimeFormatter.ofPattern("MMMM-YY");  
// ↳ January 2020
```

- MMMM → January
- MMM → Jan
- MM → 01
- M → 1
- dd → jour du mois 14
- yyyy → 2018
- yy → 18
- hh → heure
- mm → minute

- **Conversion d'une chaîne de caractère en date ou/et heure**

```
DateTimeFormatter customFormat=DateTimeFormatter.ofPattern("MM dd yyyy");  
LocalDate d=LocalDate.parse("01 02 2015",customFormat);  
LocalTime t=LocalTime.parse("11:22");  
// ↳ si le format n'est pas spécifié, utilise celui par défaut
```

# Méthodes et Encapsulation



# Déclaration de méthode

```
public final void nap(int minutes) throws Exception {  
}
```

<b>public</b>	<b>Modificateur d'accès</b>	
<b>final</b>	<b>Spécificateur optionnel</b>	
<b>void</b>	<b>Type de retour</b>	obligatoire
nap	<b>Nom de méthode</b>	obligatoire
( <b>int</b> minutes)	<b>Liste de paramètres</b>	obligatoire au minimum ( )
<b>throws</b> Exception	<b>Liste d'exceptions</b>	
{ }	<b>Corps de la méthode</b>	obligatoire au minimum { }

**Attention, à l'ordre dans la déclaration**

# Déclaration de méthode



- **Appel de fonction**

`nap(10);`

- **Modificateur d'accès**

**public** la méthode peut être appelée par toutes les classes

**private** la méthode peut être appelée uniquement à l'intérieur de la même classe

**protected** la méthode peut être appelée par une classe dans le même package ou dans une sous-classe

pas de modificateur (default), la méthode peut être appelée par une classe dans le même package



# Déclaration de méthode



- **Spécificateur optionnel** (Il peut en avoir de 0 à plusieurs)  
static, abstract, final
- **Type de retour**  
Le type de retour est obligatoire, s'il n'y a pas de valeur de retour, on utilise void

## Attention :

- Il faut vérifier que le corps de la méthode contienne une instruction return  
Si le type de retour est void, on peut avoir return; ou l'instruction return peut être omise
- Au type de valeur retournée

```
String test(){  
    return 42.0;    // Ne compile pas  
}
```

# Déclaration de méthode

- **Nom de la méthode**  
même règle que pour les variables
- **Liste de paramètres**  
les arguments sont séparés par ,  
au minimum, s'il n'y a pas d'arguments → ( )
- **Liste d'exceptions**  
séparé par ,
- **Corps de la méthode**  
Au minimum { }

```
void public walk() { }  
String walk2(int a) {if(a==4) return "";}  
public void 3walk(){};  
public void walk4(int a);  
public void walk5{};
```

} Ne compile pas

# Arguments variables

## type ...nom

- Un argument variable équivaut à un tableau, mais :
  - Il n'y a qu'un seul paramètre variable par méthode
  - Il doit être en dernière position dans la liste d'arguments

```
public void walk1(int ... nums){ }  
public void walk2(int n, int ... nums){ }  
public void walk3(int ... nums, string str){ } // ne compile pas  
public void walk4(int ... nums, string ... str){ } // ne compile pas
```

- On peut appeler une méthode avec un paramètre variable, en
  - passant un tableau
  - listant les éléments du tableau et en laissant java le créer
  - ne passant pas d'argument (java crée un tableau de taille 0)

```
walk2(1, new int[]{4,5}); // nums.length → 2  n→1 nums→{4,5}  
walk2(5,7);              // nums.length → 2  n→2 nums→ {5,7}  
walk1();                  // nums.length → 0  
walk1(null);              // NullPointerException
```

# Méthodes et attributs static



- **Méthode static** → méthode de classe partagée par toutes les instances de la classe, pas besoin d'instancier l'objet pour appeler la méthode
- **Attribut static** → état partagé par toutes les instances de la classe

```
public class Commande {  
    public static int count=0;  
    public static void main(String [] args){ // ... }  
}
```

- On peut appeler une variable et une méthode static, à partir :
  - du nom de la classe → `Commande.count`
  - d'une instance d'un objet (même si null)

```
Commande c = new Commande();  
c.count = 0;  
Commande c2 = null;  
c2.count = 5;
```

# Méthodes et attributs static



- Les variables `static` sont initialisées avec la valeur par défaut du type
- Dans une méthode `static`, on ne peut pas appeler une méthode ou variable d'instance sans instancier un objet

```
public class User {  
    private String name = "hello";  
  
    public static void main(String[] args) {  
        System.out.println(name); // Ne compile pas  
    }  
}
```

# Bloc d'initialisation static

```
static {  
  
}
```

- Un bloc d'initialisation statique est un bloc d'initialisation exécuté **uniquement à la première utilisation de la classe**

```
private static int one;  
private static final int two;  
private static final int three = 3;  
private static final int four; // Ne compile pas  
  
static {  
    one = 1;  
    two = 2;  
    three = 3; // Ne compile pas  
    two = 4;   // Ne compile pas  
}
```

# static import

- `static import` → importe **les membres** static d'une classe

```
import static java.util.Arrays.asList;  
// plus besoin d'Arrays pour appeler la méthode  
List<String> list= asList("one", "two");
```

**Attention**, si dans une classe, on définit aussi une méthode **static asList**, elle aura la priorité sur la méthode importée

```
import static java.util.Arrays;  
// ↳ ne compile pas: uniquement des membres de la classe  
import static java.util.Arrays.asList;  
import java.util.Arrays;  
static import java.util.Arrays.*;    // Ne compile pas  
public class BadStaticImports {  
    public static void main(String[] args) {  
        Arrays.asList("one"); // Ne compile pas  
        // ↳ (la classe Arrays n'est pas importée)  
    }  
}
```

# Passage de paramètre

- En java, les données sont passées par valeur  
↳ la méthode reçoit une copie de la variable

```
class Test {  
    public static void calc(int a) {  
        a = 10;  
        System.out.println(a);  
    }  
    public static void main(String[] args) {  
        int i=42;  
        Test.calc(i);           // affiche 10  
        System.out.println(i); // affiche 42  
    }  
}
```



# Passage de paramètre

## Attention avec les références

```
class Test {  
    public static void suffixe(StringBuffer sb, StringBuffer sa) {  
        sb.append("suffixe");  
        // ↳ la référence n'est pas modifiée, mais l'état de  
        //    l'objet va l'être  
        sa = new StringBuffer("modif");  
        // ↳ la référence est modifiée  
        System.out.println(sa);  
        System.out.println(sb);  
    }  
  
    public static void main(String args[]) {  
        StringBuffer str1 = new StringBuffer("aaa");  
        StringBuffer str2 = new StringBuffer("bbb");  
        Test.suffixe(str1, str2);    // affiche aaasuffixe modif  
        System.out.println(str1);    // affiche aaasuffixe  
        System.out.println(str2);    // affiche bbb  
    }  
}
```

# Passage de paramètre et surcharge de méthodes



- **Surcharge de méthodes**

surcharge de méthode → signature de méthode différente avec le même nom et des paramètres de différent type

```
public boolean fly(){ };  
public void fly(short i) { };  
public void fly(int i) { };  
public int fly(int i) { }; // ne compile pas  
public static void fly(int i) { }; // ne compile pas
```

- **Surcharge et Autoboxing**

L'autoboxing aura lieu uniquement, s'il n'y a pas de méthode avec un paramètre de type primitif

```
void fly(int num){ } // fly(3) → appel toujours fly(int)  
void fly(Integer num) { }
```

# Passage de paramètre et surcharge de méthodes



- **Surcharge et Varargs**

```
public void fly(int[] lengths) {};  
public void fly(int... lengths) {}; // ne compile pas  
// varargs équivaut à un tableau
```

- **Ordre de choix de la bonne méthode surchargée**

1) Type correspondant parfaitement

```
public String glide(int i, int j) { }
```

2) Type primitive plus grand

```
public String glide(long i, long j) { }
```

3) Type "autoboxing"

```
public String glide(Integer i, Integer j) { }
```

4) Argument variable

```
public String glide(int... nums) { }
```

# Constructeur

- Nom du constructeur = nom de la classe
- pas de type de retour

```
public class Bunny
    public Bunny() { }
    public bunny() { }           // ne compile pas
    public void Bunny() { }     // compile, mais ce n'est pas
                                // un constructeur (type de retour)
}
```

- Le constructeur est utilisé quand on crée un nouvel objet  
↳ instantiation `new Bunny()`;
- **Constructeur par défaut**
  - s'il n'y a pas de constructeur  
↳ java en crée un par défaut (sans argument)
  - dès que l'on définit un constructeur dans la classe  
↳ il n'est plus fourni par java

- **Constructeur privé**

Si l'on définit un constructeur privé, java ne fournit plus de constructeur par défaut et l'on ne peut plus instancier l'objet

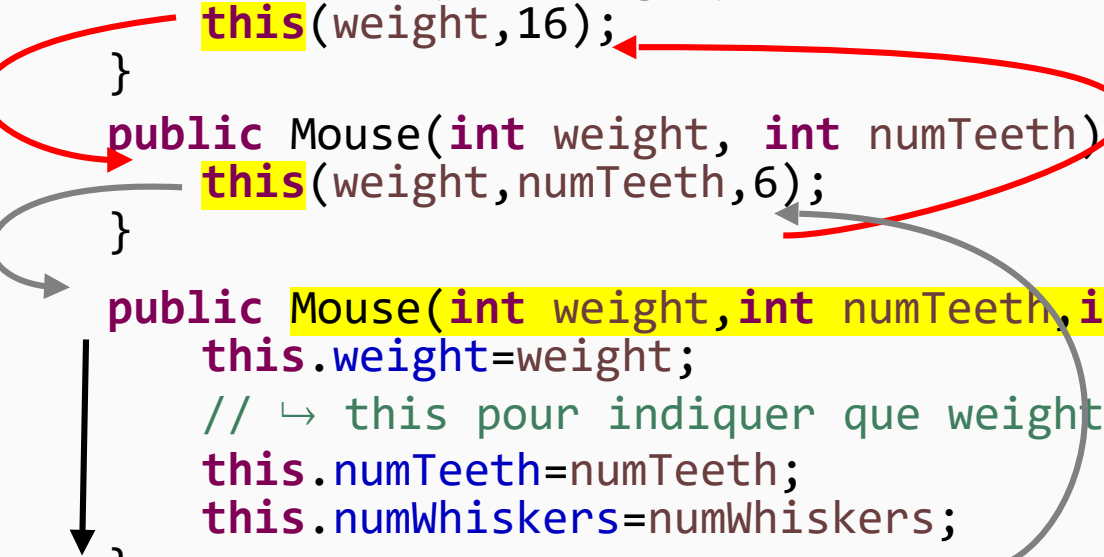
↳ utile lorsqu'une

- la classe n'a que des méthodes static
- la classe veut contrôler tous les appels pour créer des instances elle-même

# Surcharge du constructeur

- `this` → permet de chaîner l'appel des constructeurs  
this doit être la première ligne du constructeur

```
public class Mouse{  
    private int numTeeth;  
    private int numWhiskers;  
    private int weight;  
    public Mouse(int weight){  
        this(weight,16);  
    }  
    public Mouse(int weight, int numTeeth){  
        this(weight,numTeeth,6);  
    }  
    public Mouse(int weight,int numTeeth,int numWhiskers){  
        this.weight=weight;  
        // ↳ this pour indiquer que weight est une variable d'instance  
        this.numTeeth=numTeeth;  
        this.numWhiskers=numWhiskers;  
    }  
    //...  
}
```



# Attribut final

- Un attribut **final** ne peut être initialisé qu'une seule fois
- Le constructeur peut initialiser un attribut **final**, s'il n'a pas été initialisé

```
public class MouseHouse {  
    private final int volume;  
    private final String name="The mouse house";  
  
    public MouseHouse(int length, int width, int height){  
        volume=length*width*height;  
        name= "Other name" ;    // ne compile pas  
                                // name a déjà été initialisé  
    }  
}
```

# Ordre d'initialisation

- 1) S'il y a une super-classe, elle est initialisée d'abord
- 2) Déclarations des variables static et des blocs d'initialisation static
- 3) Déclaration des attributs d'instance et des blocs d'initialisations dans l'ordre où il apparaisse dans le fichier
- 4) Le constructeur

```
public class InitOrder {  
    private String name="Marcel";           // 4 name ← Marcel  
    {  
        System.out.println(name);          // 5 affiche name  
    }                                       // 1 COUNT ← 0  
    private static int COUNT = 0 ;          // 2 affiche COUNT  
    static{System.out.println(COUNT) ;}     // 3 COUNT ← 10 et l'affiche  
    static {COUNT+=10 ; System.out.println(COUNT) ;}  
    public InitOrder(){  
        System.out.println("constructeur") ; // 6 affiche constructeur  
    }  
}  
public class CallInitOrder {  
    public static void main(String args[]){ InitOrder init= new InitOrder(); }  
}
```



# Encapsulation



- Règle pour les javaBeans à connaître pour l'examen :
  - **attributs privés**  
`private int a;`  
`private boolean b;`
  - **méthodes getter**  
`private int getA() { return a; }`  
`private boolean isB() { return b; }`
  - **méthodes setter**  
`private void setA(int a){ this.a = a; }`  
`private void setB(boolean b){ this.b = b; }`
  - Le nom des méthodes doit avoir un préfixe **set**, **get** ou **is** (pour les booleans) suivit du nom de la propriété, dont la première lettre est en majuscule

# Classe immuable

- Une classe immuable n'a pas de setter et on fixe les valeurs des attributs avec le constructeur
- Elles ne peuvent plus être changées après l'instanciation

```
public class ImmutableSwan {  
    private int numberEggs;  
  
    public ImmutableSwan(int numberEggs) {  
        this.numberEggs = numberEggs;  
    }  
  
    public int getNumbersEggs() {  
        return numberEggs;  
    }  
}
```

# Expression lambda

- Une expression lambda peut être vu comme :
  - une méthode anonyme : elle a des paramètres, un corps comme les autres méthodes mais n'a pas de nom
  - une méthode que l'on peut transmettre comme s'il s'agissait d'une variable
- **Sans expression lambda**

```
public class Animal{  
    private String species;  
    private boolean canHop;  
    private boolean canSwim;  
    public Animal(String speciesName, boolean hooper,  
        boolean swimmer) {  
        species=speciesName;  
        canHop=hopper;  
        canSwin=swimmer;  
    }  
    public boolean canHop() { return canHop;}  
    public boolean canSwim() { return canSwim;}  
    public String toString() { return species;}  
}
```

# Expression lambda

```
// Interface pour faire des tests sur animal
public interface CheckTrait {
    boolean test(Animal a);
}

// Classe pour tester si un animal peut sauter
public class CheckIfHopper implements CheckTrait {
    public boolean test(Animal a) {
        return a.canHop();
    }
}

public class Search{
    public static main(String [] args){
        List<Animal> animals=new ArrayList<Animal>() ;
        animals.add(new Animal("fish",false,true) ;
        animals.add(new Animal("kangaroo",true,false) ;
        animals.add(new Animal("rabbit",true,false) ;
        print(animals, new CheckIfHooper()) ;
    }

    public static void print(List<Animal> animals, CheckTrait checker){
        for(Animal animal : animals) {
            if(checker.test(animal)){
                System.out.println(animal + " ") ;
            }
        }
        System.out.println() ;
    }
}
```

# Expression lambda

- Si on veut tester, si un animal peut nager, il faudrait faire une classe `CheckIfSwimmer` qui implémente l'interface `CheckTrait`
- **Avec expression lambda**

Avec une expression lambda, on n'a plus besoin des classes `CheckIfHopper` et `CheckIfSwimmer`

On peut remplacer dans le main:

```
print(animals, new CheckIfHooper());
```

par

```
print(animals, a-> a.canHop());
```

si on affiche tous les animaux qui peuvent nager

```
print(animals, a-> a.canSwim());
```

ou qu'ils ne peuvent pas nager

```
print(animals, a-> !a.canSwim());
```

# Syntaxe des expressions lambda



`a -> a.canHop();`

Diagram illustrating the syntax of a lambda expression:

- `a` is labeled as the **nom du paramètre** (parameter name).
- `canHop()` is labeled as the **corps de la méthode** (method body).

- La méthode prend un paramètre animal et retourne un booléen
- On passe l'expression lambda comme second paramètre de print qui est un CheckTrait. Java essaye de "mapper" le lambda à l'interface CheckTrait qui a pour unique méthode :

**boolean** test(Animal a)

↳ donc, l'expression lambda a pour paramètre un Animal et retourne un boolean

# Syntaxe incluant les parties optionnelles

nom du paramètre

corps de la méthode

```
(Animal a) -> { return a.canHop(); }
```

Obligatoire, s'il y a un bloc

type du paramètre (optionnel)

- Les parenthèses peuvent être omises uniquement, s'il n'y a qu'un seul paramètre et que le type n'est pas défini
- On peut supprimer les accolades s'il n'y a qu'une instruction
- Quand il n'a pas d'accolade, java n'a pas besoin de type de retour ni de point virgule

# Syntaxe incluant les parties optionnelles



```
print(()-> true) ;  
print(a-> a.startsWith("test")) ;  
print((String a) a->a.startsWith("test")) ;  
print((a,b)-> a.startsWith("test")) ;  
print((String a, String b) a->a.startsWith("test")) ;  
  
(a,b) -> {int a=0 ; return 5;}  
// ↳ ne compile pas car avec int a on redéfini la  
// variable locale a  
  
(a,b) -> {int c=5 ; return 5;} // OK
```



# Predicates



- Les expressions lambda fonctionnent avec une interface contenant une seule méthode, comme **CheckTrait**. pour ne pas avoir à en définir Java fourni un interface **Predicate** dans le package `java.util.function` :

```
public interface Predicate<T> {  
    boolean test(T t) ;  
}
```

- avec la méthode **print** cela donne :

```
public static void print(List<Animal> animals, Predicate<Animal> checker)  
{  
    // ...  
}
```

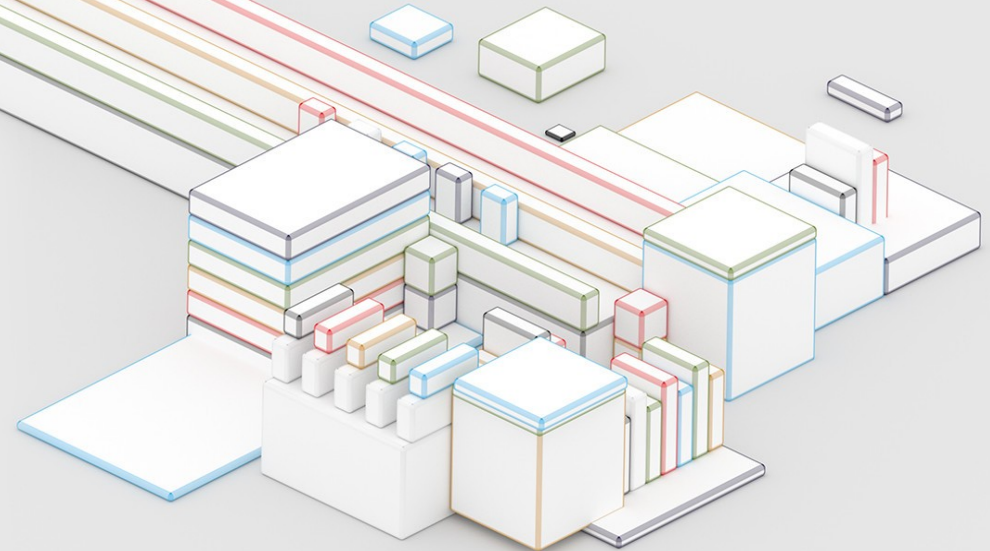
# Predicates

- Dans Java 8 l'interface **Predicate** est intégré dans des classes existantes. Pour la certification, on doit ne connaître que la méthode **removeIf** de **ArrayList**

```
List<String> text = new ArrayList<>();  
text.add("AAA");  
text.add("BBB");  
text.add("CCC");  
text.add("DDD");  
text.add("EEE");  
text.removeIf(s -> s.charAt(0) != 'C');  
// ↳ text ne contient plus que CCC
```

- Pour **removeIf** un **predicate** est définie, il prend comme paramètre un String et retourne un booléen

# Héritage et Abstraction



# Héritage de classe

- En java, :
  - On ne peut faire que de l'héritage simple
  - Tous les classes héritent de la classe Object
- Pour qu'une classe hérite d'une autre, on doit ajouter le mot-clé **extends** suivie du nom de la classe parent

```
public abstract ElephantSeal extends Seal { }
```

<b>public</b>	modificateur d'accès <b>public</b> ou <b>default</b>	
<b>abstract</b>	<b>abstract</b> ou <b>final</b>	optionnel
<b>class</b>	mot-clé	
ElephantSeal	nom de la classe	optionnel
<b>extends</b> Seal	hérité de la classe parent	optionnel

# Héritage de classe



- **Modificateur d'accès**

Pour l'examen, on ne doit connaître que public et default :

- **public** : la classe est utilisable pour toutes autres les classes
- **(default)** : la classe ne peut être accessible que par les sous-classes et les classes du même package

```
class Rodent {}  
public class Groundhog extends Rodent {}
```

# Règles de définition du constructeur



- 1) La première instruction de chaque constructeur est un appel à :
  - un autre constructeur de la classe avec **this()**
  - un constructeur de la classe parente directe avec **super()**
- 2) **super()** ne peut pas être appelé après la première instruction du constructeur
- 3) Si aucun appel à **super()** n'est déclaré dans un constructeur, Java insérera un **super()** sans argument comme première instruction du constructeur
- 4) Si le parent n'a pas de constructeur sans argument et que l'enfant ne définit aucun constructeur, le compilateur lancera une erreur et essaiera d'insérer un constructeur par défaut dans la classe enfant

# Règles de définition du constructeur

---



- 5) Si le parent n'a pas de constructeur sans argument, le compilateur requiert un appel explicite à un constructeur parent dans chaque constructeur enfant

**Appel des constructeurs** → le constructeur parent est toujours exécuté avant le constructeur enfant

# Appel des membres de classe hérités



- Les classes Java peuvent utiliser tout membre public ou protégé de la classe parente
- Si la classe parent et la classe enfant font partie du même package, la classe enfant peut également utiliser tous les membres par défaut définis dans la classe parent
- Une classe enfant ne peut jamais accéder à un membre privé de la classe parent



# Redéfinition de méthode

- On peut redéfinir une méthode en déclarant une nouvelle méthode dans la classe enfant déclarée
- La méthode dans la classe enfant :
  - 1) doit avoir la même signature que la méthode dans la classe parent
  - 2) doit être au aussi ou plus accessible que la méthode dans la classe parent

```
class Parent {  
    public meth1(){...}  
    protected meth2(){...}  
    public meth3(){...}  
}  
  
class Enfant extends Parent {  
    public meth1(){...}           // Ok modificateur identique  
    public meth2(){...}           // Ok modificateur plus accessible  
    protected meth3(){...}        // Faux modificateur moins accessible  
}
```

# Redéfinition de méthode



- 3) ne doit pas déclarer une checked exception qui n'a pas été déclarée dans la méthode parent

```
class Parent {  
    public meth1() throws IOException {...}  
    public meth2() throws IOException {...}  
    public meth3() {...}  
}  
  
class Enfant extends Parent {  
    public meth1() throws IOException {...} // Ok identique  
    public meth2() {...} // Ok, si l'exception est  
                        // traitée dans la méthode  
    public meth3() throws IOException{...} // Faux  
}
```

- 4) si elle retourne une valeur, la valeur doit être la même ou une sous-classe que la méthode de la classe parent

# Redéclarer une méthode privée



- Il n'est pas possible de surcharger une méthode privée d'une classe parente
- Mais une méthode privée dans la classe enfant peut avoir la même signature qu'une méthode dans la classe parent. Elle sera totalement indépendante de la méthode de la classe parent

```
public class Camel(){
    private String getNumberOfHumps(){
        return " Undefined " ;
    }
}

public class BactrianCamel() extends Camel {
    private String getNumberOfHumps(){
        return "2 " ;
    }
}
```

# Cacher une méthode static

- On cache une méthode quand une classe enfant définit une méthode static avec le même nom et signature qu'une méthode static définie dans la classe parente
- En plus des 4 règles de la redéfinition des méthodes, il y a une règle supplémentaire :  
La méthode définie dans la classe enfant doit obligatoirement être static, si elle est définie static dans la classe parent

```
public class Bear {  
    public static void eat() {  
        System.out.println("Bear is eating");  
    }  
}  
  
public class Panda extends Bear {  
    public static void eat() {  
        System.out.println("Panda bear is eating");  
    }  
    // ...  
}
```

# Méthode final

- Une méthode déclarée avec final ne pourra plus être redéfinie

```
class Parent {  
    public final boolean test() {  
        return true;  
    }  
}  
  
class Enfant extends Parent {  
    public boolean test() { // Ne compile pas : on ne peut  
        return false;      // pas redéfinir une méthode  
    }                       // final  
}
```

# Règles de définition des classes abstraites



- 1) Elles ne peuvent pas être instanciées directement
- 2) Elles peuvent définir ou pas des méthodes abstraites et non abstraites
- 3) Elles ne peuvent pas être **private**, **protected** ou **final**
- 4) Une classe abstraite qui étend une autre classe abstraite hérite de toutes les méthodes abstraites, comme si c'était les siennes
- 5) La première classe concrète doit fournir une implémentation pour toutes les méthodes abstraites héritées

# Règles de définition des classes abstraites

```
public abstract class Animal {  
    private int age;  
  
    public void eat() {  
        System.out.println(" animal is eating ");  
    }  
  
    public abstract String getName();  
}  
  
public class Swan extends Animal {  
    public String getName() {  
        return "Swan";  
    }  
}
```

# Règles de définition des méthodes abstraites



- 1) Elles ne sont définies uniquement dans une classe abstraite
- 2) Elles ne peuvent pas être **private** ou **final**
- 3) Pas d'implémentation / de corps{ } dans la classe abstraite
- 4) L'implémentation des méthodes abstraites dans une sous-classe suivent les mêmes règles que la redéfinition de méthode



# Interfaces

```
public abstract interface CanBurrow {  
    public static final int MINIMUM_DEPTH = 2;  
    public abstract int getMaximumDepth();  
}
```

Pas nécessaire de les ajouter,  
Le compilateur le fera  
automatiquement

<b>public</b>	Modificateur d'accès public ou default
<b>interface</b>	Mot clé pour définir une interface
CanBurrow	Nom de l'interface

## Règles pour définir une interface

- 1) Une interface ne peut être instanciée directement
- 2) Une interface n'a pas obligatoirement de méthode
- 3) Une interface ne peut pas être **final**, **private** ou **protected**
- 4) Les méthodes d'une interface ne peuvent pas être **final**, **private** ou **protected**

# Interfaces

```
public interface CanFly {  
    void fly(int speed);           // abstrait et public,  
    abstract void takeoff();       // sont supposés  
    public abstract double dive(); // si on les fournis  
    // ou pas, le compilateur les insérera automatiquement  
}  
  
private final interface CanCrawl {  
    private void dig(int depth);  
    protected abstract double depth();  
    public final void surface();  
}
```

- **Implémentation d'une interface**

Une classe peut implémenter plusieurs interfaces, chacune séparée par ,

```
public class Elephant implements WalksOnFourLegs, HasTrunk, Herbivore {  
}
```

# Hériter d'une interface



- Une interface qui étend une autre interface, hérite de toutes les méthodes abstraites comme ses propres méthodes abstraites
- La première classe concrète qui implémente une interface, ou étend une classe abstraite qui implémente une interface, doit fournir une implémentation pour toutes les méthodes abstraites héritées

```
public interface HasTail {  
    public int getTailLength();  
}  
  
public interface HasWhiskers {  
    public int getNumberOfWhiskers();  
}  
  
public interface Seal extends HasTail, HasWhiskers {  
}
```

# Hériter d'une interface



- Si une classe implémente deux interfaces qui contiennent la même méthodes :
  - si elles ont la même signature: pas de conflit, si l'on implémente l'une des deux méthodes, on implémente automatiquement la deuxième méthode
  - si elles n'ont pas la même signature: pas conflit, considéré comme une surcharge de la méthode
  - si elles ont la même signatures avec un type de retour différent: ne compile pas

# Variable d'interface

- 1) Les variables d'interface sont supposées être **public**, **static** et **final**
- 2) La valeur d'une variable d'interface doit être définie lors de sa déclaration (**final**)

```
public interface CanSwim {  
    int MAXIMUM_DEPTH = 100;  
    final static boolean UNDERWATER = true;  
    public static final String TYPE = "Submersible";  
}  
  
public interface CanDig {  
    private int MAXIMUM_DEPTH = 100;  
    protected abstract boolean UNDERWATER = false;  
    public static String TYPE;  
}
```

# Méthode d'interface par défaut



- Une méthode par défaut dans une interface définit une méthode abstraite avec une implémentation par défaut
  - La classe qui implémente l'interface peut remplacer la méthode par défaut. Si elle ne le fait pas l'implémentation par défaut est utilisée
  - La méthode par défaut permet aux développeurs d'ajouter de nouvelles méthodes à une interface sans casser les implémentations existantes de cette interface
- 1) Une méthode par défaut ne peut être déclarée que dans une interface
  - 2) Une méthode par défaut doit être marquée avec le mot-clé par défaut. Dans ce cas on doit fournir un corps de méthode

# Méthode d'interface par défaut



- 3) Une méthode par défaut ne peut pas être static, final ou abstract
- 4) Comme toutes les méthodes d'une interface, une méthode par défaut est supposée être public
- Quand une interface hérite d'une autre interface qui contient une méthode par défaut, il peut:
  - choisir d'ignorer la méthode par défaut
    - ↳ l'implémentation par défaut sera utilisée
  - redéfinir la méthode par défaut
  - redéclarer la méthode comme abstraite

# Méthode d'interface par défaut

```
public interface HasFins {  
    public default int getNumberOfFins() {  
        return 4;  
    }  
    public default double getLongestFinLength() {  
        return 20.0;  
    }  
    public default boolean doFinsHaveScales() {  
        return true;  
    }  
}  
  
public interface SharkFamily extends HasFins {  
    public default int getNumberOfFins() {  
        return 8;  
    }  
    public double getLongestFinLength(); // force la classe  
    // qui implémentera l'interface à redéfinir la méthode  
    public boolean doFinsHaveScales() { // Ne compile pas  
        return false;                 // il manque default  
    }  
}
```



# Méthode d'interface par défaut



- Si une classe implémente deux interfaces qui ont des méthodes par défaut avec le même nom et la même signature, le compilateur générera une erreur  
Sauf si la sous-classe remplace les méthodes par défaut en double et supprime l'ambiguïté

```
public interface Walk {  
    public default int getSpeed() {  
        return 5;  
    }  
}  
  
public interface Run {  
    public default int getSpeed() {  
        return 10;  
    }  
}
```

```
public class Cat implements Walk, Run {  
    public int getSpeed() {  
        return 1;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(new Cat().getSpeed());  
    }  
}
```

# Méthode static d'interface

Les méthode statiques d'une interface sont identiques aux méthodes statiques défini dans les classes. Mais elles ne sont héritées d'aucune classe qui implémente l'interface

- 1) Comme toutes les méthodes d'une interface, une méthode statique est supposée être public
- 2) Pour référencer la méthode statique, une référence au nom de l'interface doit être utilisée

```
public interface Hop {  
    static int getJumpHeight() {  
        return 8;  
    }  
}  
  
public class Bunny implements Hop {  
    public void printDetails() {  
        System.out.println(Hop.getJumpHeight());  
        System.out.println(getJumpHeight()); // Ne compile pas  
    }  
}
```

# Polymorphisme



- Un objet Java est accessible à l'aide :
  - d'une référence du même type que l'objet
  - d'une référence qui est une superclasse de l'objet
  - d'une référence qui définit une interface que l'objet implémente (directement ou via une superclasse)
- **Objet vs Référence**
  - 1) Le type de l'objet détermine quelles propriétés existent dans l'objet en mémoire
  - 2) Le type de la référence à l'objet détermine quelles méthodes et variables sont accessibles au programme Java
- **Casting Objects**
  - 1) La conversion d'un objet d'une sous-classe en une superclasse ne nécessite pas de conversion explicite

- 2) La conversion d'un objet d'une superclasse en une sous-classe nécessite une conversion explicite
- 3) Le compilateur n'autorisera pas les transtypages vers des types non liés
- 4) Même lorsque le code compile sans problème, une exception peut être levée au moment de l'exécution si l'objet en cours de conversion n'est pas réellement une instance de cette classe

## Méthodes virtuelles

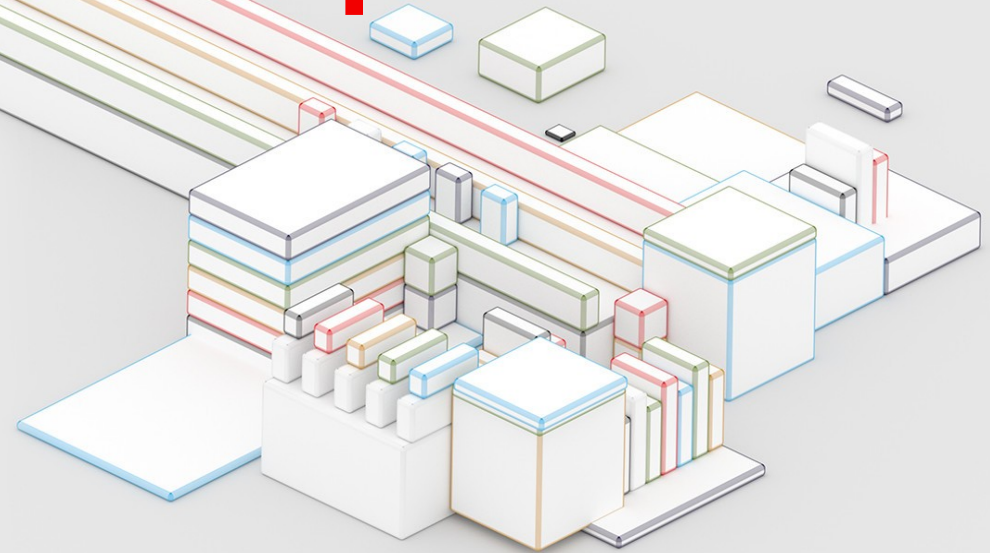
- Une méthode virtuelle est une méthode dans laquelle l'implémentation spécifique n'est déterminée qu'au moment de l'exécution

# Polymorphisme



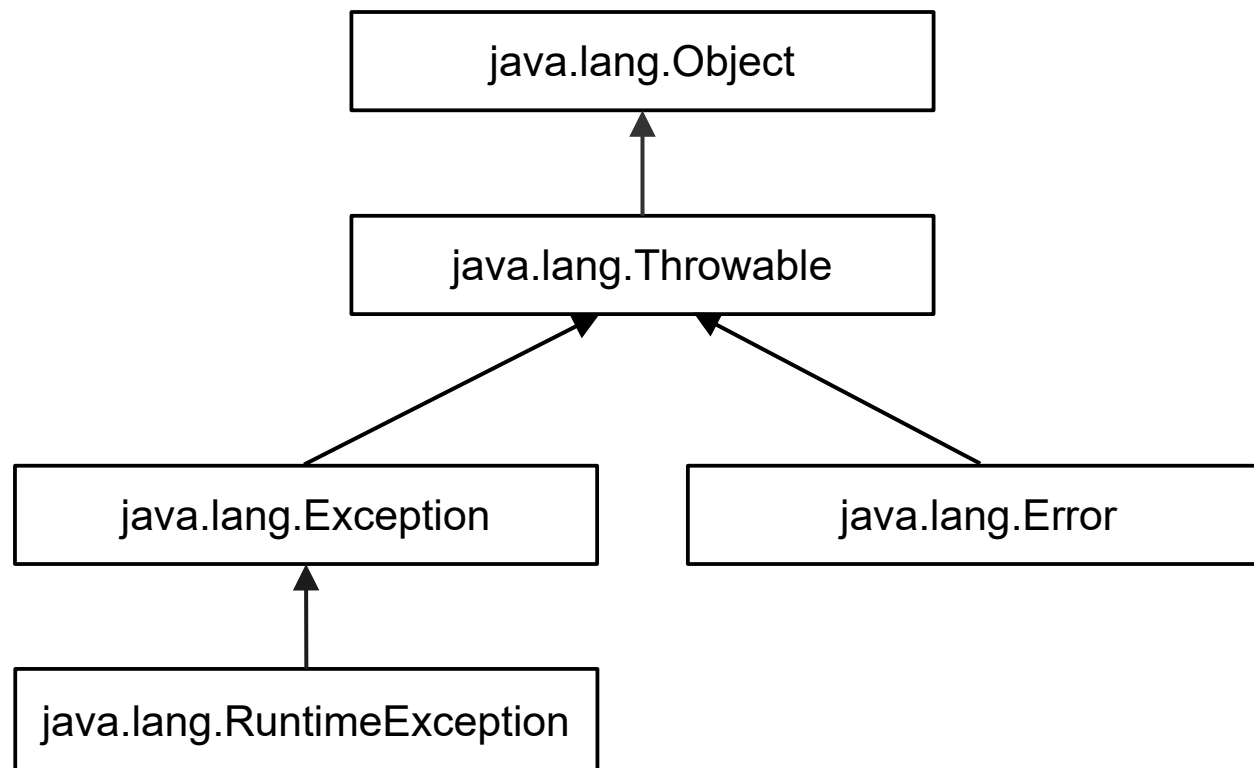
- Si l'on appelle une méthode sur un objet qui remplace une méthode, on obtient la méthode substituée, même si le l'appel à la méthode se trouve sur une référence parent ou dans la classe parent
- Toutes les méthodes qui ne sont pas final, static ou private sont considérées comme des méthodes virtuelles

# Gestion des exceptions



# Définition

Situations inattendues ou exceptionnelles qui surviennent pendant l'exécution d'un programme, interrompant le flux normal d'exécution



# Types d'exceptions



- **Errors**

On ne doit pas les identifier et le programme s'arrête en les rencontrant

Exemple : La JVM charge une classe inexistant

- **Runtime exceptions**

Ne peuvent être prévues (dans certains cas)

Exemple : Essayer de lire une valeur en dehors d'un tableau

- **Checked exceptions**

Le développeur doit obligatoirement les anticiper et coder des lignes pour les traiter

↳ **Règle java: handle** (try / catch) ou **declare** (throws en fin de la signature de la méthode)

Exemple : Ouvrir un fichier qui n'existe pas



# Lancer une Exception



Dans l'examen, il y a deux types de code qui ont pour résultat une exception:

- Une erreur dans le code

```
String[] str = new String[5];  
System.out.println(str[7]); // ArrayIndexOutOfBoundsException
```

- Explicitement avec Le mot-clé **throw**, qui est utilisé pour déclencher une exception

```
throw new Exception();  
throw new RuntimeException();  
throw new Exception("message");
```

# Bloc try et catch

- Utilisé pour encadrer un bloc de code susceptible de déclencher une exception

Si une exception est déclenchée dans le bloc **try** et qu'elle est listée dans le **catch**, le code qui se trouve dans le bloc catch va être exécuté

```
try {  
    // des lignes de code susceptibles de lever  
    // une exception  
}  
catch (type_exception e) {  
    // capture et traitement de l'exception de type :  
    // type_exception  
    // e fait référence à l'objet exception capturé  
}
```

# Bloc try et catch

## Attention, au code non atteignable

```
try{  
    // ...  
}  
catch(Exception e){  
    // ...  
} catch( IOException e){  
    // Ne compile pas, car ce bloc ne peut pas être atteint  
    // le premier catch intercepte toutes les exceptions même  
    // les IOException  
}  
  
try{  
    // ...  
}  
catch(IOException e){  
    // ...  
} catch( Exception e){  
    // ok  
}
```

# Bloc finally

- Le bloc **finally** est toujours exécuté, qu'une exception soit déclenché dans le bloc try ou pas

```
try {  
    // ...  
}  
catch (IOException e) {  
    // ...  
}  
finally {  
    // Toujours exécuté avec ou sans exception  
}
```

## Attention :

- Respecter l'ordre : **try** → **catch** → **finally**
- Si un **System.exit(0)** est appelé dans le **try** ou dans le **catch**, le bloc **finally** n'est pas exécuté

# Les exceptions à connaître



## Runtime Exception

- **ArithmeticException**  
↳ division par 0
- **ArrayIndexOutOfBoundsException**  
↳ index illégal pour accéder à un tableau
- **ClassCastException**  
↳ Caster un objet en une sous-classe, dont il n'est pas l'instance
- **IllegalArgumentException**  
↳ une méthode a reçue un argument illégal ou inapproprié
- **NullPointerException**  
↳ Conversion d'une chaîne en nombre avec une chaîne invalide

# Les exceptions à connaître



## Checked Exceptions

- **FileNotFoundException**
  - ↳ le code essaie de référencer un fichier qui n'existe pas
- **IOException**
  - ↳ problème de lecture ou d'écriture d'un fichier

## Errors

- **ExceptionInInitializerError**
  - ↳ une exception est lancée dans un bloc d'initialisation static
- **StackOverflowError**
  - ↳ lancé par l'exécution d'une méthode récursive sans condition de sortie
- **NoClassDefFoundError**
  - ↳ L'erreur se produit quand java ne peut pas trouver la classe à l'exécution



**Plus d'informations sur <http://www.dawan.fr>**

**Contactez notre service commercial au  
09.72.37.73.73 (prix d'un appel local)**