

Introduction à Java et JUnit

Sommaire

- ❖ **Qu'est-ce que JAVA ?**
- ❖ **La notion d'objet**
- ❖ **L'environnement de développement**
- ❖ **Les classes**
 - Les variables d'instances
 - Les types de données
 - Le constructeur
- ❖ **Les méthodes**
 - Les paramètres
 - Les types de retour
 - Les instructions
 - Les modificateurs d'accessibilité
 - Les accesseurs
- ❖ **Ecrire des tests en JAVA**
- ❖ **Concepts de la programmation orientée objet**
 - Encapsulation
 - Héritage
 - Interface
 - Polymorphisme
 - Enumération



Sommaire

❖ Sélections et décisions

- if... else
- switch...case

❖ Tableaux et boucles

- tableaux
- for
- while

❖ Collections

- Les types de Collection
- Typage des éléments d'une collection
- Parcours d'une collection

❖ Exceptions



Qu'est-ce que Java?

- Java est un langage de programmation mis au point par Sun Microsystems dans les années 90.
- La première version sort en 1995



Qu'est-ce que Java?

- Un langage de programmation objet basé sur la notion de classe (les objets sont décrits/regroupés dans des classes)
- Langage sûr, parce que fortement « typé »
 - Mots réservés (class, package, if, else, while, int, void, ...)
 - Syntaxe (déclaration de variable => *type nom_variable; .* Par ex: **int a;**)
 - Normes de codage et de nommage
- Java fournit également une librairie standard de classes (API)
- Les applications Java ne sont pas compilées en langage machine mais en bytecode. Ce bytecode nécessite un programme spécial appelé (JVM – Java Virtual Machine) pour être interprété.



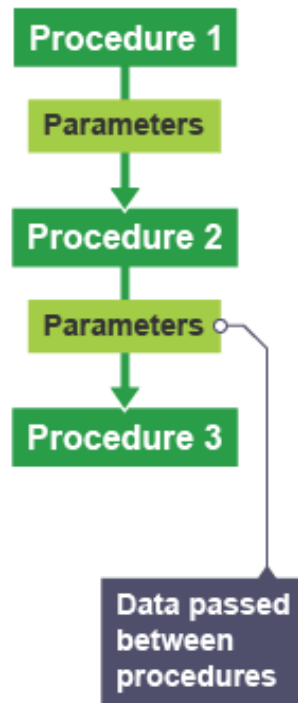
Vous avez dit « orienté objet » ?

Programmation procédurale

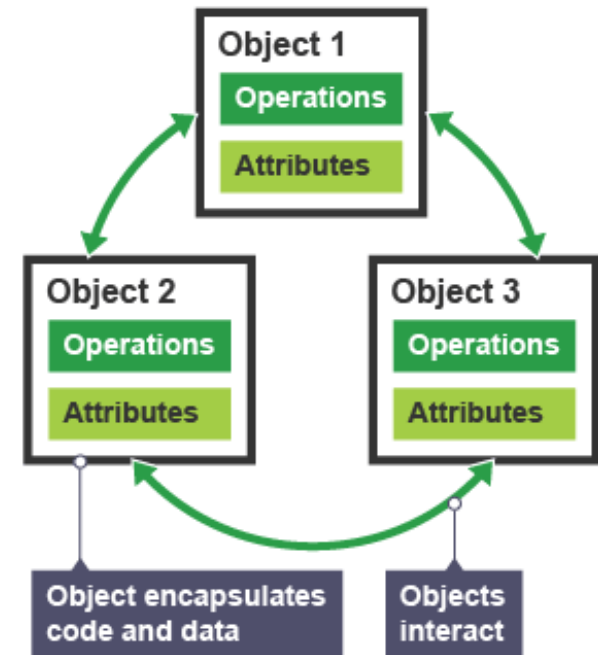
VS

Programmation orientée objet

Procedural language



Object-oriented language



CODE



DONNÉES

Vous avez dit « orienté objet » ?

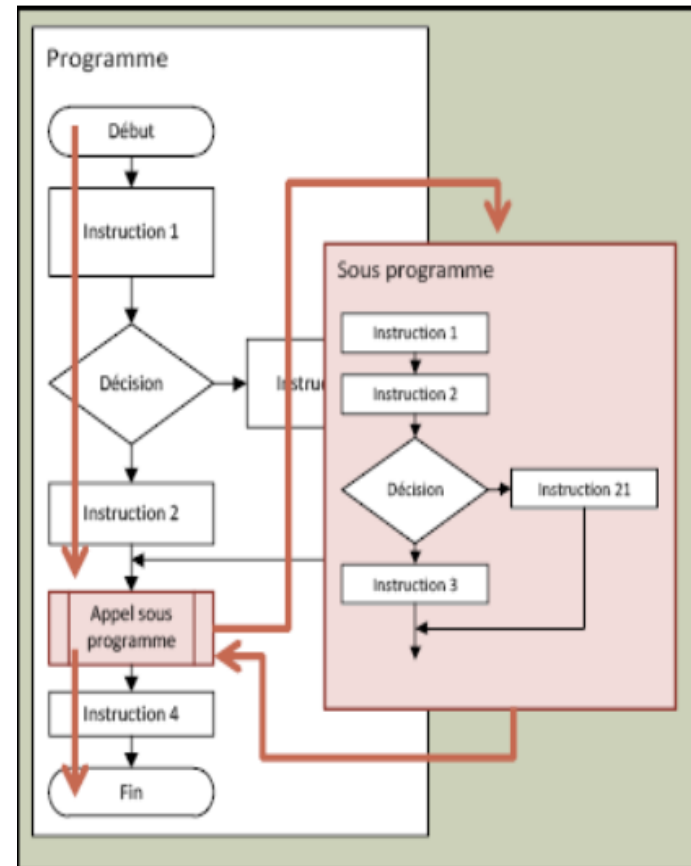
La programmation procédurale

- Une suite d'instructions s'exécutant les unes après les autres
- Avec des procédures ou des fonctions (sous-programmes)

Cette approche permet de décomposer les fonctionnalités d'un programme en procédures qui s'exécutent séquentiellement.

Inconvénients

- Le **procédural** est très éloigné de notre manière de penser
- Le **développeur doit penser de manière algorithmique** (proche du langage de la machine).
- Le **code** est **peu lisible** => Et **difficile** à **modifier**, à **maintenir**
- L'**ajout** de fonctionnalités **difficile** => Réutilisation du code incertaine
- Le **travail d'équipe** est **délicat**



Vous avez dit « orienté objet » ?

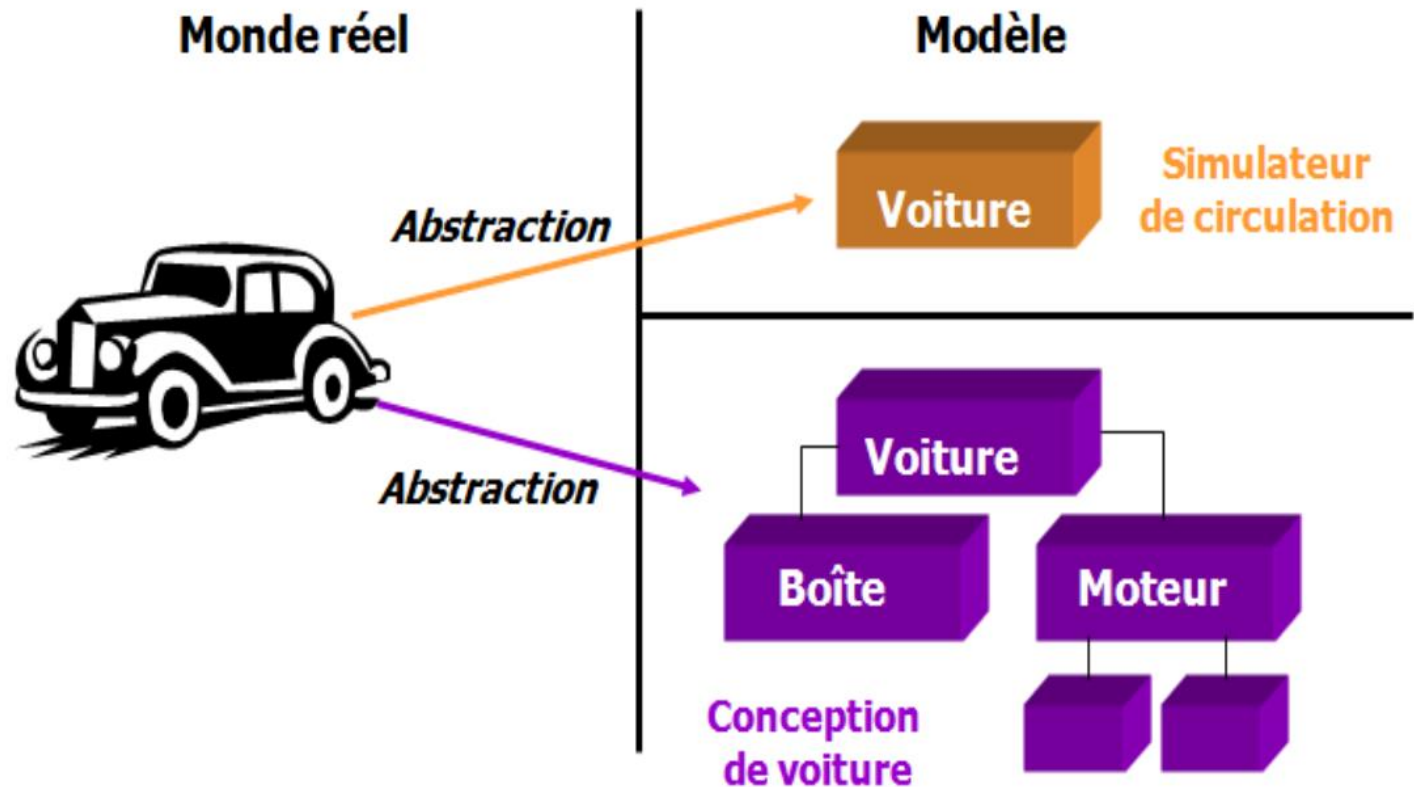
La programmation orientée objet

- Finalement, il est peut-être plus simple de **s'inspirer du monde réel**
 - **Le monde réel est composé d'objets**, d'êtres vivants, de matière
 - Pourquoi ne pas programmer de manière plus réaliste ?
- **Les objets ont des propriétés, des attributs**
 - Un chat a 4 pattes, un serpent aucune
- **Les objets ont une utilité, une ou plusieurs fonctions = des méthodes**
 - Une voiture permet de se déplacer



Qu'est-ce qu'un objet ?

Un objet est l'abstraction d'une entité du monde réel



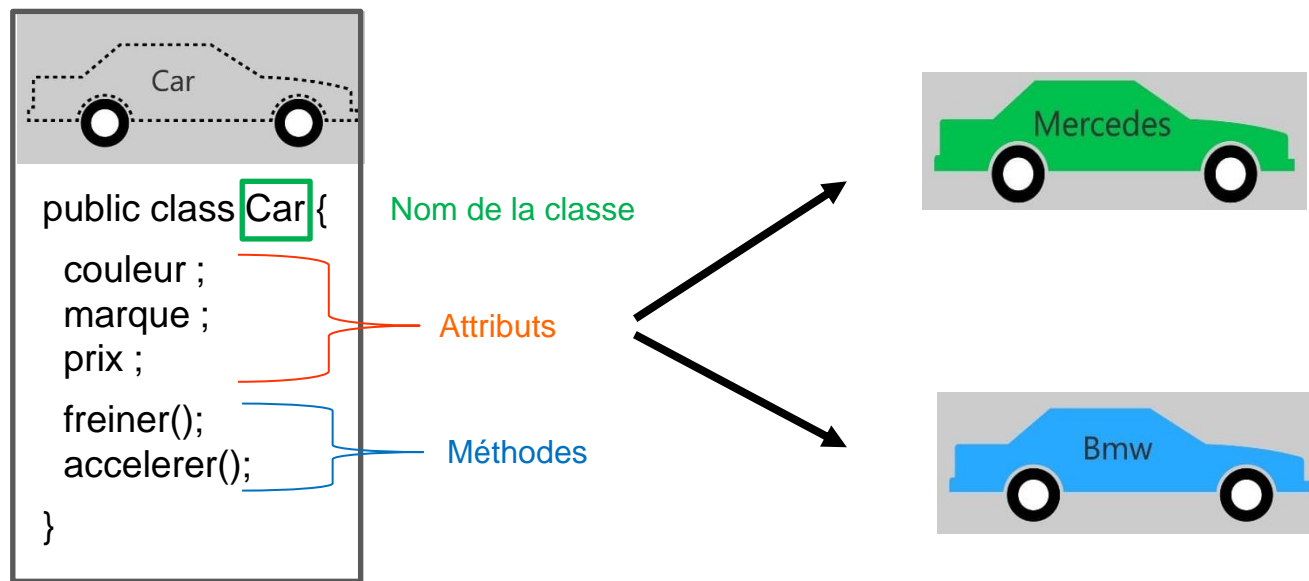
Qu'est-ce qu'un objet ?

- **Un élément qui modélise toute entité, concrète ou abstraite, manipulée par le logiciel...**
 - un élément avec un état interne donné par des valeurs **d'attributs** appelées aussi « **variables internes** »
 - *Exemple : L'objet voiture peut avoir plusieurs attributs : 4 roues, un volant, un moteur, etc*
- **Un élément qui réagit à certains messages qu'on lui envoie de l'extérieur**
 - C'est son comportement, ce qu'il sait faire : **on parlera de méthodes**
 - *Exemple : Avec cette voiture, je peux tourner, accélérer, freiner, ...*
- **Un élément qui ne réagit pas toujours de la même manière**
 - Son comportement dépend de l'état dans lequel il se trouve. **On parlera d'instance**
 - *Exemple : Essayez de démarrer sans essence !!?*



Des classes aux objets

- Avant de créer des objets, il faut définir un modèle
- Des objets pourront être créés à partir de ce modèle
- Ce modèle s'appelle **une classe**
- Les objets fabriqués à partir du modèle sont **des instances**



Notion de classe

- Un programme Java est composé de classes (mot clé: **class**)
- Tout le code permettant d'exécuter des instructions se trouve dans des *classes*

Nom du package

{

```
package fr.eql.autom.java;
```

Corps de classe avec

- (des attributs)
- (des méthodes)

{

```
class PremiereClasse {  
  
}
```

En-tête de classe avec

- le mot réservé class
- le nom de la classe
- (un modificateur d'accessibilité)
- (des annotations)

- Le nom d'une *classe* débute par une majuscule et est écrit en **CamelCase**
- Une *classe* appartient à un **package**. Les *packages* permettent de ranger les *classes* (comme une suite de dossiers)
- Le nom d'un package est une suite de caractères en minuscule séparés par des points « . »

Notion de méthode

Une classe contient des méthodes. Les *méthodes* sont des commandes qu'il est possible d'appeler afin d'exécuter des instructions.

corps de méthode avec :
• (des instructions)

```
package fr.eql.autom.java;

class DeuxiemeClasse {

    public void disBonjour() {

        System.out.println("Bonjour!");

    }

}
```

En-tête de méthode avec :

- (un modifieur d'accessibilité)
- un type de résultat/retour
- un nom
- des () contenant les paramètres séparés par des virgules

Instruction
(ici un appel de la méthode `println(String arg0) : void`)

- Le nom des *méthodes* débute par une minuscule et s'écrit en camelCase.
- La *méthode* `disBonjour()` permet d'écrire « Bonjour! » dans la console.
- Une *instruction* Java se termine par un point-virgule « ; »

Exercice

L'IDE Eclipse

- 1/ Ouvrir Eclipse
- 2/ Créer un projet Maven
- 3/ Ouvrir le fichier pom.xml



Apache Maven – A quoi sert Maven?

Maven est un outil qui permet de gérer la production des logiciels Java ... de l'écriture du code au déploiement d'un exécutable.

- Il introduit des conventions dans la structure du projet Java
 - src/main/java
 - src/test/java
 - src/main/ressources
 - src/test/ressources
 - target
- Il permet de gérer les dépendances de code
- Il permet de gérer les dépôts de dépendances
- Il définit un cycle de vie logiciel
 - Compile
 - Test
 - Package
 - Install
 - deploy



Maven – pom.xml d'un projet Selenium WebDriver / JUnit

POM = Project Object Model

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.eql.autom</groupId>
  <artifactId>simple</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
  </dependencies>
</project>
```

JUnit est un framework de tests unitaires standard en Java
Parallèlement à notre développement java, nous créerons des tests pour vérifier la qualité de notre code

Exercice 1

Créer sa première classe

- 1/ Dans le répertoire « main/java », créer une nouvelle classe « Eleve » appartenant au package « eql.java ».
- 2/ Ecrire la méthode publique « apprendre » (vide)



Créer sa première classe

... et maintenant ?

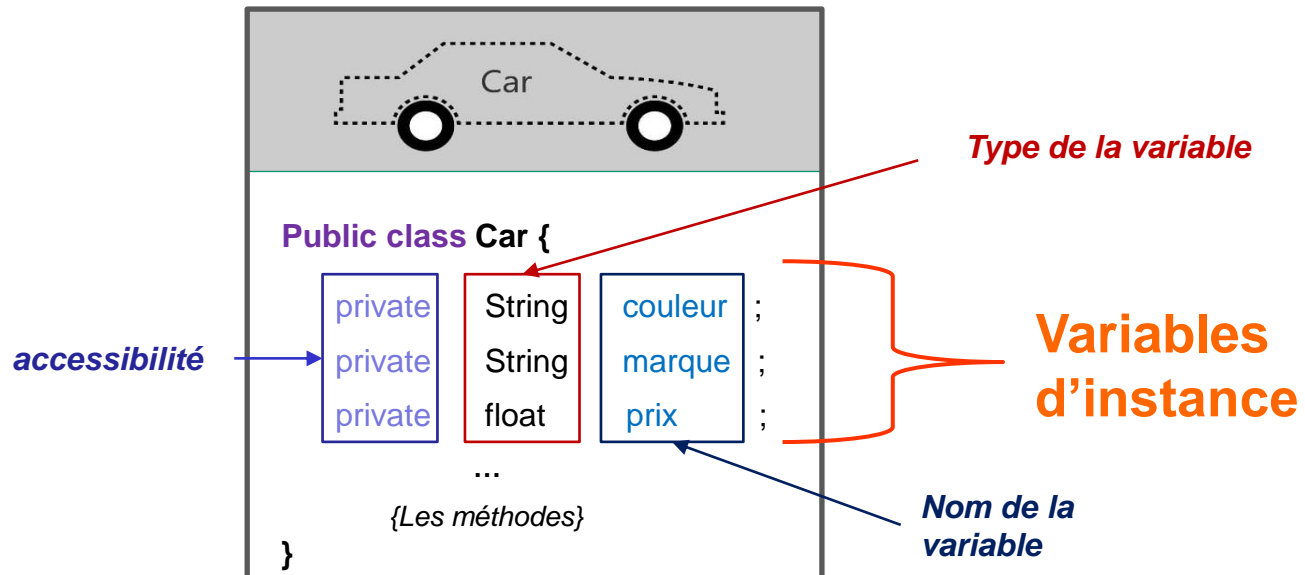
- Nous avons une **classe** (Eleve), pour laquelle nous avons défini une **méthode** : `apprendre()`;
- Pour autant, ce modèle est assez pauvre...
- L'idée est que, à partir du modèle « Eleve », nous puissions générer différentes instances d'un – ou « d'une »!! – élève.
- Pour cela, deux éléments manquent à notre classe :
 - Des **variables d'instance**, a partir desquelles sera dérivée l'état de notre instance
 - Un **constructeur**, qui nous permettra de générer des instances



Les variables d'instance

Quels sont les attributs de mon objet ?

- Les variables d'instance seront déclarées dans le corps de la classe (au début, de préférence...).
- Leur création répond à la question « quels sont les attributs de mon objet ? »
→ ex : **une voiture** sera caractérisée par **sa marque, sa couleur, son modèle, son prix, etc ...**
- Comme pour n'importe quelle autre variable, on déclare une variable d'instance en déclarant son **type** et son nom.
- Les variables d'instances ont la particularité d'être définie en tant qu'élément « **private** » (on verra l'utilité de ce mot réservé plus tard...)



Les types de données

En Java, toute donnée se voit attribuer un type

Parmi les données on distingue :

- Les objets, qui ont pour type la classe qui les définit
- Les données de type primitif, qui appartiennent à l'un des huit types primitifs

Les huit types primitifs

Booléens

- **boolean** (true /false)

Caractères

- **char** (« b »)

Entiers

- **byte** (de -128 à 127)
- **short** (de -32768 à 32767)
- **int** (de -2147483648 à 2147483647)
- **long** (de -9223372036854775808 à 9223372036854775807)

Nombres à virgule

- **float** (précision de 7 chiffres après la virgule)
- **double** (précision de 15 chiffres après la virgule)

String n'est pas un type primitif !

String est une classe, utilisée comme type représentant une chaîne de caractères. Ce n'est pas un type primitif mais c'est un type spécial

Exercice 2

Définir des variables d'instance

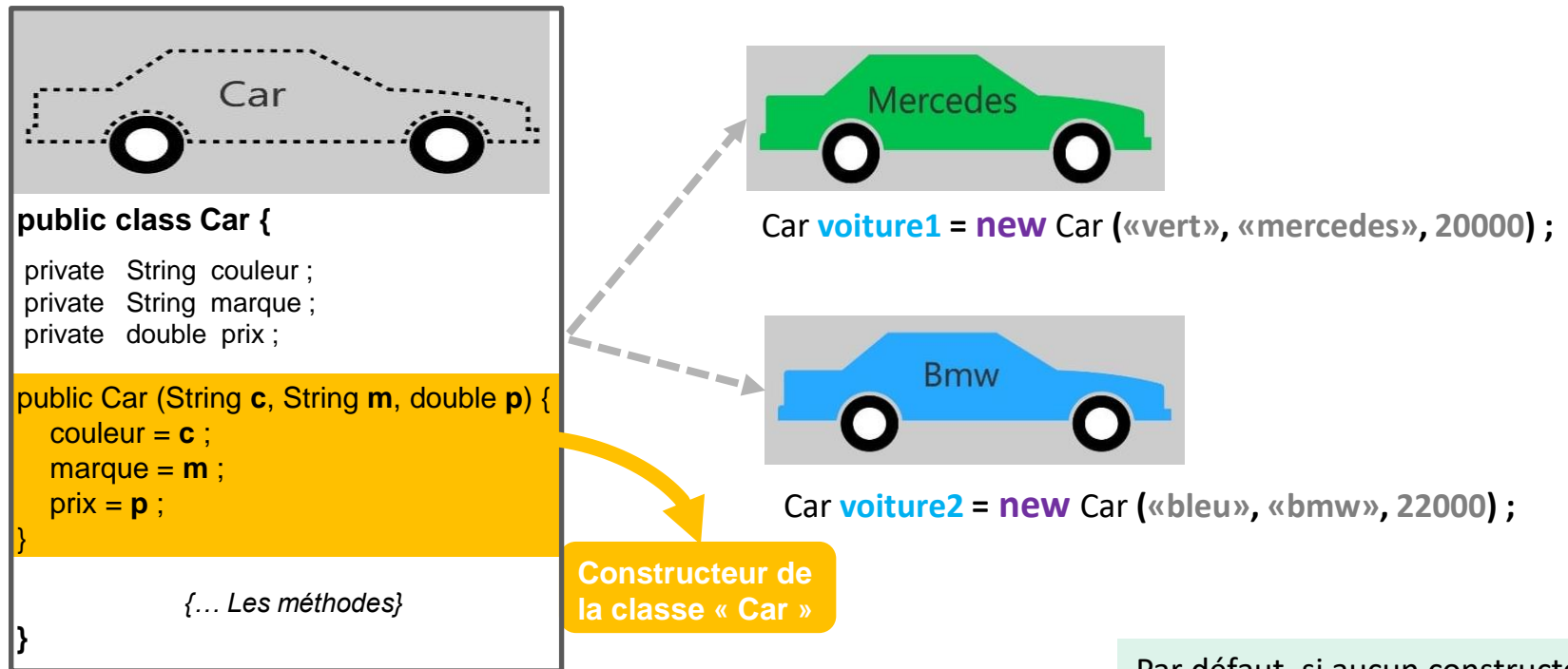
- 1/ Définir des variables d'instance pour la classe Eleve.
- 2/ Définir le type auquel rattacher chacune des variables.



Le constructeur

Créer des instances de ma classe

- Une classe a un ou plusieurs constructeurs qui servent à créer les instances et à initialiser leur état
- Un constructeur a **le même nom** que la classe et n'a pas de type retour



L'instance créée :

- **A son propre état interne** (les valeurs des variables d'instance) .
- **Partage le code qui détermine son comportement** (les méthodes) avec les autres instances de la classe.


Par défaut, si aucun constructeur n'est défini explicitement dans une classe, il existe toujours un constructeur sans arguments.

Le constructeur

Utilisation du « **this** »

Il est parfois commode d'utiliser le même nom pour les variables d'instance et les paramètres du constructeur.

Dans un tel cas, il est alors nécessaire de rendre le code explicite et non-ambigu, en utilisant le mot clé **this** pour **désigner l'attribut**.



```
public class Car {  
    private String couleur ;  
    private String marque ;  
    private double prix ;  
  
    public Car (String couleur, String marque, double prix) {  
        this.couleur = couleur ;  
        this.marque = marque ;  
        this.prix = prix ;  
    }  
  
    {... Les méthodes}  
}
```

Exercice 3

Ecrire un constructeur

A partir des variables d'instance de la classe Eleve, écrire un constructeur .



Les méthodes

Les différents éléments d'une méthode



```
public class Car {
```

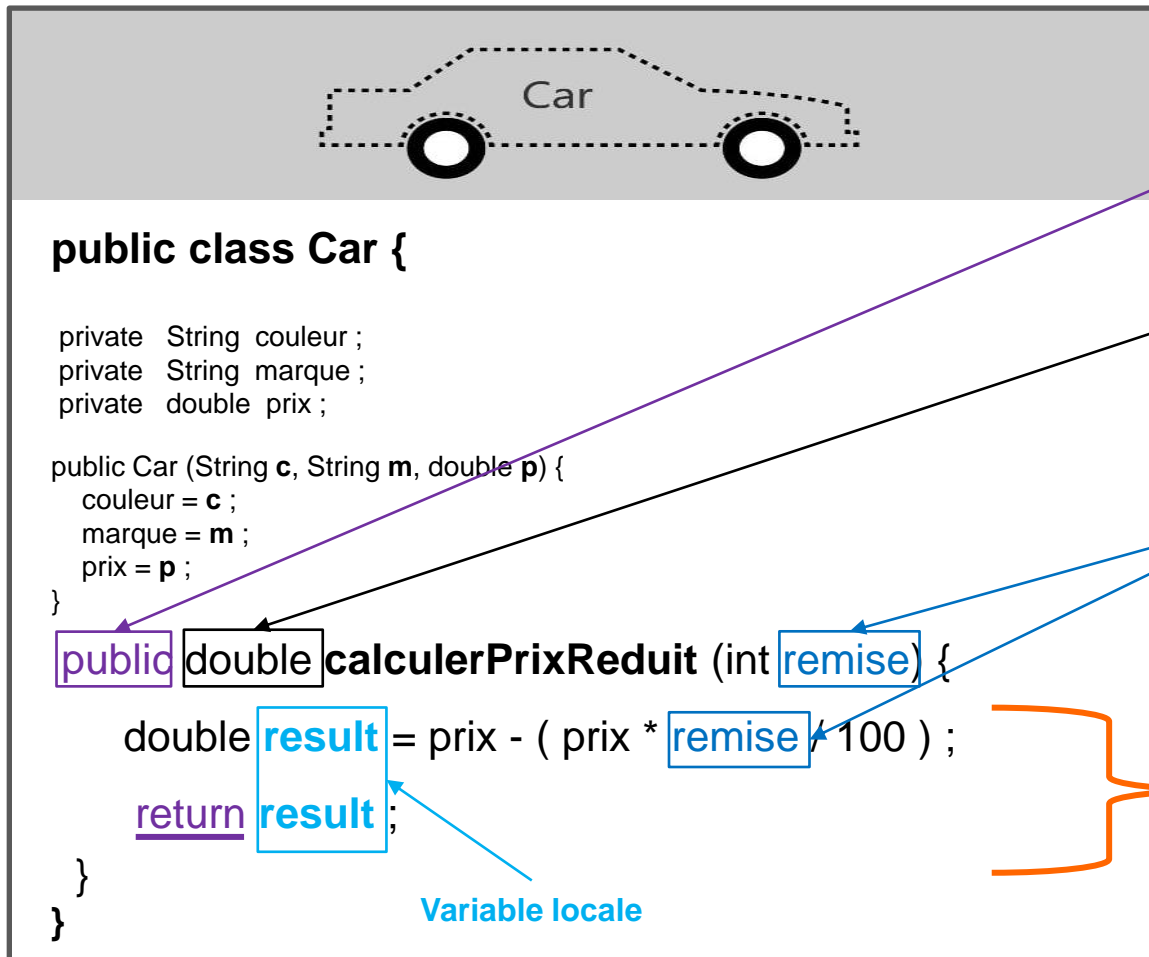
```
    private String couleur ;  
    private String marque ;  
    private double prix ;
```

```
    public Car (String c, String m, double p) {  
        couleur = c ;  
        marque = m ;  
        prix = p ;  
    }
```

```
    public double calculerPrixReduit (int remise) {  
        double result = prix - ( prix * remise / 100 ) ;  
        return result ;  
    }  
}
```

Les méthodes

Les différents éléments d'une méthode



**Modifieur
d'accessibilité**

**Type de retour
de la méthode**

**Paramètre de la
méthode**

**corps de méthode avec
des instructions :**

- Une déclaration de la variable locale 'result' de type double (+ affectation d'une valeur a partir d'un calcul)
- Le retour de la méthode (mot réserver **return**)

Les méthodes

Les paramètres d'une méthode

- Souvent, les méthodes (comme les constructeurs) ont besoin qu'on leur passe **des données initiales sous la forme de paramètres**
- Le type des paramètres doit être indiqué dans la déclaration de la méthode

Ex 1

```
public int addition (int a, int b) {  
    return a + b ;  
}
```

Ex 2

```
public String passerEnMinuscule (String phrase) {  
    return phrase.toLowerCase() ;  
}
```

- Quand la méthode ou le constructeur n'a pas de paramètre, on ne met rien entre les parenthèses

Ex 3

```
public void disBonjour () {  
    System.out.println(« Bonjour ») ;  
}
```

Les méthodes

Le type de retour d'une méthode

- Quand la méthode renvoie une valeur, on doit indiquer le type de la valeur renvoyée dans la déclaration de la méthode

Ex 1

```
public int addition (int a, int b) {  
    return a + b ;  
}
```

Ex 2

```
public String passerEnMinuscule (String phrase) {  
    return phrase.toLowerCase() ;  
}
```

Ex 3

```
public void disWhatever (String text) {  
    System.out.println(text) ;  
}
```

Les méthodes

Méthodes de la classe String

- `length()` : **int**
retourne le nombre de caractères dans une chaîne
- `isEmpty()` : **boolean**
retourne vrai si la chaîne est de longueur 0
- `equals(String str)` : **boolean**
- `substring(int fromIndex)` : **String**
- `substring(int fromIndex, int endIndex)` : **String**
- `toLowerCase()` : **String**
- `toUpperCase()` : **String**
- `trim()` : **String**

... quelques exemples

```
public int afficherNombreLettre(String mot) {  
    return mot.length();  
}
```

```
public boolean isVariableNonAffecte(String var) {  
    return this.var.isEmpty();  
}
```

```
public boolean verifierMessageBienvenu(String msg) {  
    return msg.substring(0, 6).equals("Welcome");  
}
```

```
public String passerEnMajuscule(String phrase) {  
    return phrase.toUpperCase();  
}
```

Opérateurs dans le langage Java

Assignation (=)

- `int i = 0;`

Arithmétique (+ - * / ++ --)

- `int j = 1 + 1;`
- `int k = 1 - 1;`
- `int m = 2 * 2;`
- `int n = 2 / 1;`
- `n++;`
- `m--;`

Opérations sur les chaînes (+)

- `String var = "de" + "but";`

Comparaison (> < >= <= == !=)

- `boolean res1 = 1 > 0;`
- `boolean res2 = 1 < 1;`
- `boolean res3 = 1 >= 1;`
- `boolean res4 = 1 <= 0;`
- `boolean res5 = 1 == 0;`
- `boolean res6 = 1 != 0;`

Logique (&& || !)

- `boolean res7 = true && false`
- `boolean res8 = true || false`
- `boolean res9 = ! true`

Grands types d'instructions

- **déclaration de variables**
 - `int i;`
 - `String s;`
 - `Voiture v;`
- **affectation d'une valeur à une variable**
 - `i = 2;`
 - `s = "bonjour"; s = s.toUpperCase();`
 - `v = new Voiture(); v = contrat. acheterVoiture();`
- **instanciation d'objet**
 - `new Voiture("Ford");`
- **appel de méthode**
 - `s.length();`

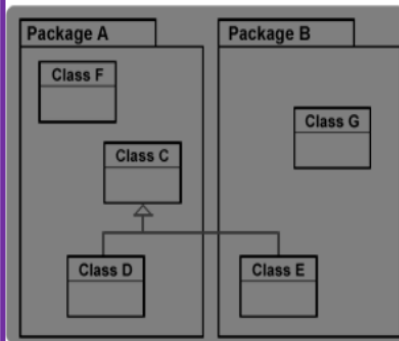


Les méthodes

Les modificateurs d'accessibilité

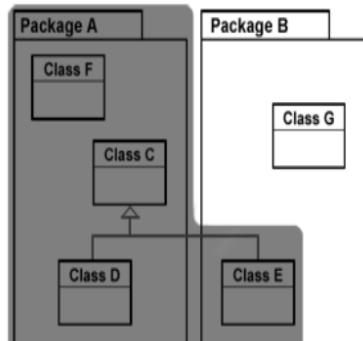
- Toutes les méthodes et données définies au sein d'une classe sont utilisables par toutes les méthodes de la classe.
- Lors de la conception d'une classe, il faut décider des méthodes/variables qui seront visibles à l'extérieur de cette classe (principe d'encapsulation).
- Java implémente la protection des 4 P (**public**, **package**, **protected**, **private**)

public



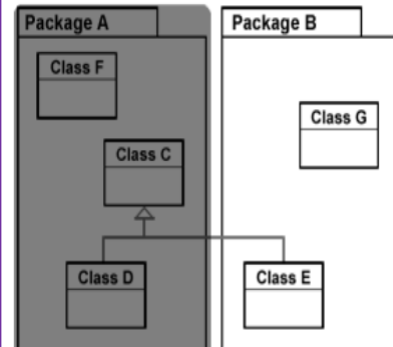
Visible depuis
n'importe quelle classe,
n'importe quel package

protected



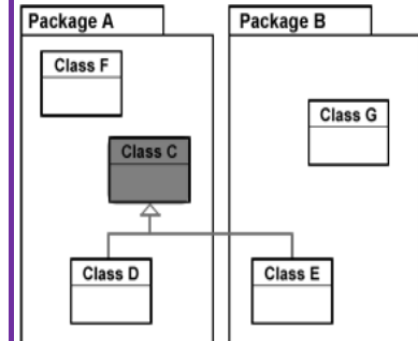
visible uniquement
dans le package et dans
les classes *dérivées** de
cette classe

Par défaut *package**



Visibilité par défaut
pour toutes les classes au
sein d'un même package
– **pas de mot clé**

private

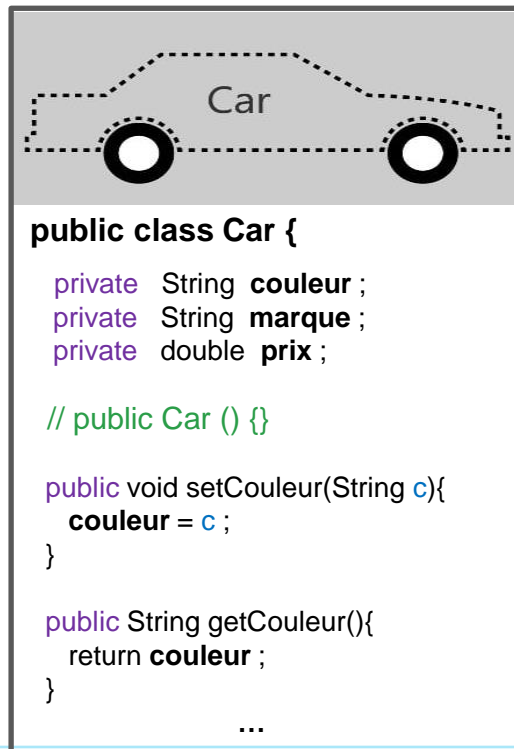


visible uniquement au
sein de la classe

Les méthodes

Les accesseurs (*getter* et *setter*)

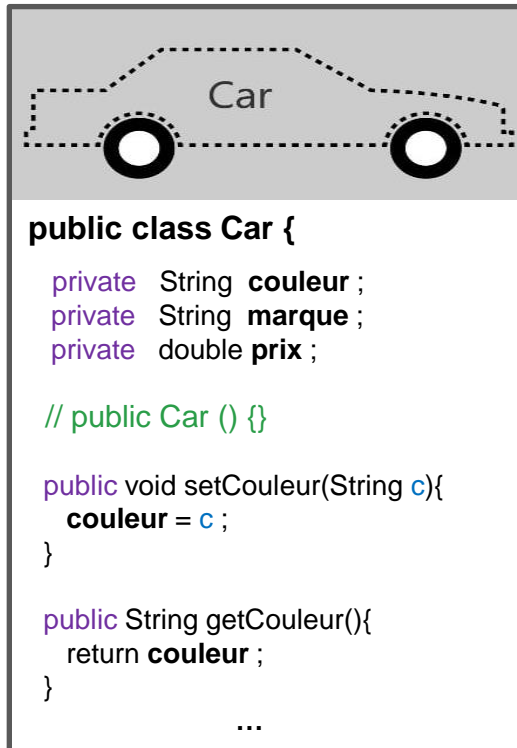
- Deux types de méthodes servent à donner accès aux variables d'instance (**private**) depuis l'extérieur de la classe
 - ❖ Les **accesseurs en lecture** pour lire les valeurs des variables → **getter** en anglais
 - ❖ Les **accesseurs en écriture** pour modifier leur valeur → **setter** en anglais



```
public class UneAutreClasse {  
    ...  
  
    Car voiture1 = new Car () ;  
    voiture1.setCouleur(« bleue »);  
    voiture1.setMarque(« Bmw »);  
    System.out.println(« Quelle belle voiture » + voiture1.getCouleur());  
    ...  
}
```

Les méthodes

Les accesseurs (*getter et setter*)



```
// création de l'instance voiture1 à partir du constructeur implicite de la classe Car  
Car voiture1 = new Car () ;  
  
// appel de la fonction setCouleur(String) de la classe Car  
voiture1.setCouleur(« bleue »);  
  
// appel de la fonction setMarque(String) de la classe Car  
voiture1.setMarque(« Bmw »);  
  
// affichage en console de la concaténation de « Quelle belle voiture » et du  
// résultat de la méthode getCouleur de la classe « Car » renvoyant la valeur de la  
// variable 'couleur'  
System.out.println(« Quelle belle voiture » + voiture1.getCouleur();
```

Console

Quelle belle voiture bleue

Exercice 4

Implémenter des méthodes accesseurs

Pour la classe « Eleve » :

1/ Créer deux variables d'instance en renseignant leur type respectif :

- « absent » (qui ne peut prendre comme valeur que vrai ou faux)
- « niveau_classe » (qui détermine le niveau de la classe de l'élève)

2/ Implémenter les méthodes getter et setter pour chacune de ces variables



JUnit

Ecrire des tests en Java

- Java peut être utilisé pour écrire des tests.
- **JUnit** est un framework de tests unitaires standard en Java.
- *JUnit* contient en plus des instructions Java habituelles des instructions de validation appelées **assertions**.
- La version la plus récente de JUnit est JUnit 5... nous utiliserons **JUnit 4**, notamment caractérisé par la présence d'**annotations**.

```
package fr.eql.autom.java;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class DeuxiemeTest {

    @Test
    public void peutComparerDeuxChainesDeCaracteres() {
        String string1 = "Hello"+"World";
        assertEquals("Hello+World=HelloWord", "HelloWorld", string1);
    }
}
```

Diagram illustrating the structure of a JUnit test class with annotations and imports:

- package**: `fr.eql.autom.java;`
- imports**: `import org.junit.Test;` and `import static org.junit.Assert.assertEquals;`
- classe**: `public class DeuxiemeTest {`
- annotation**: `@Test`
- Méthodes avec instructions**: `public void peutComparerDeuxChainesDeCaracteres() {`
- assertion**: `assertEquals("Hello+World=HelloWord", "HelloWorld", string1);`

Méthode de vérification JUnit

- **assert ...** (*[message]*, *value*) type retour **boolean**

Permet de contrôler une propriété

assertTrue	→	Vérifie qu'une condition est vraie
assertFalse	→	Vérifie qu'une condition est fausse
assertNull	→	Vérifie qu'une variable est nulle
assertNotNull	→	Vérifie qu'une variable est non nulle

- **assert ...** (*[message]*, *expectedValue*, *actualValue*) type retour **boolean**

Permet de contrôler une propriété

assertEquals	→	Compare les valeurs de 2 variables
assertNotEquals	→	Compare les valeurs de 2 variables
assertArrayEquals	→	Compare le contenu de 2 tableaux
assertSame	→	Compare les références de 2 objets
assertNotSame	→	Compare les références de 2 objets

- **fail()** type retour **void**

Permet de déclencher un échec du test

Exercice 5

Faire un premier test unitaire

1/ Dans le répertoire « test/java », créez une classe de test appelée « MaClasseDeTest » (appartenant au package de votre choix...)

2/ En utilisant les méthodes `setAbsent()` et `getAbsent()`, ainsi que les méthodes `assert` de Junit, faites un test qui :

- Instancie un objet `Eleve` (que vous appellerez « `eleve` »)
- Valorise la variable « `absent` » (`true` ou `false`, selon votre choix...)
- Affiche en console la valeur de la variable « `absent` »
- Vérifie (teste!) que la valeur de la variable `absent` est bien celle attendue...

[aidez-vous des slides 34, 36 et 37]



Les méthodes et les mots-clés

Méthodes et propriétés statique

- Le mot-clé *static* permet de déclarer une méthode – ou une propriété – qui n'est pas liée à une instance de classe mais à la classe elle-même.
- Cela signifie que ces éléments ne sont pas dans le contexte d'un objet lorsqu'ils sont utilisés mais dans le contexte de la classe.
- On les appelle en utilisant le nom de la classe et non le nom d'une instance (nom de variable).



Les méthodes et les mots-clés

Exemple de méthode statique



```
public class Car {
```

```
    private String couleur ;  
    private String marque ;  
    private float  prix ;
```

```
    public Car (String c, String m, float p) {  
        couleur = c ;  
        marque = m ;  
        prix = p ;  
    }
```

```
    public double calculerPrixRduit (int remise) {  
        double result = prix - ( prix * remise / 100 ) ;  
        return result ;  
    }
```

Cette méthode est dynamique, puisqu'elle dépend de la valorisation de la variable **prix** par une instance de classe

```
    public static double calculerAcceleration (float vitesselnit, float vitessFinal, float duree) {  
        double result = (vitessFinal - vitesselnit) / duree  
        return result ;  
    }
```

Cette méthode est '**static**', puisqu'elle n'est liée à aucune instance de la classe.

Les méthodes et les mots-clés

... d'autres mot-clés

- **final** (classes, méthodes et propriétés)
 - Sert à déclarer que l'élément qu'il marque est immuable.
 - S'il s'agit d'une classe, il n'est pas possible de la dériver.
 - S'il s'agit d'une méthode, il n'est pas possible de la redéfinir / la surcharger.
 - S'il s'agit d'une propriété, sa valeur ne peut pas être modifiée.
- **abstract** (classes et méthodes)
 - Sert à déclarer qu'un élément n'est pas complètement défini.
 - Une classe peut être déclarée comme abstraite.
 - Elle ne peut alors pas être instanciée
 - Elle peut en lieu et place d'une méthode contenir une simple déclaration de signature de méthode portant elle-même le mot-clé **abstract**



Exercice 6

... continuer le développement

- 1/ Créez trois nouvelles classes : « Ecole », « Enseignant » et « PersonnelAdministratif ».
- 2/ Sachant que la classe « Ecole » restera immuable, quel mot-clef pouvez-vous lui attribuer ?
- 3/ Pour les classes « Enseignant » et « PersonnelAdministratif », créez :
 - Quatre variables d'instances « nom », « prenom », « salaire », « nb_absences_mois_en_cours ».
 - Un constructeur avec deux paramètres qui renseigneraient les variables « nom » et « prenom » à l'instanciation.
 - Les méthodes getter et setter pour les variables « salaires » et « nb_absences_mois_en_cours ».
 - Une méthode « sePresenter » qui affiche en console une présentation de l'enseignant ou personnel administratif incluant son nom, prenom , son poste et salaire.
- 4/ Dans une classe JUnit, testez vos méthodes *getter* et *setter* ainsi que la méthode « sePresenter ».
- 5/ Dans la classe « Ecole », créez une méthode permettant d'augmenter le salaire d'un employé de l'école. TESTEZ CETTE MÉTHODE!



Héritage

Le principe

- **Objectif** : raccourcir les temps d'écriture et de mise au point d'une application en réutilisant le code déjà implémenté. (mais aussi : éviter copier-coller -> maintenabilité)
- **La méthode** : réunir des objets possédant des caractéristiques communes dans une nouvelle classe, plus générale, appelée « super-classe ».
- On parle alors d'**héritage**
- En Java, **chaque classe a une et une seule classe mère** (pas d'héritage multiple), dont elle hérite les variables et les méthodes
- Le mot clé est **extends**.



Héritage

Hérédité et surcharge des méthodes

?

```
public class Vehicule {  
    private int nb_roues ;  
  
    public Vehicule (int nb_roues) {  
        this.nb_roues = nb_roues ;  
    }  
  
    public void avancer() {  
        ...  
    }  
}
```

*La classe « Car » peut surcharger
le constructeur de « Vehicule »
grâce au mot clé **super()**.*



*La classe « Car » hérite
de la classe « Vehicule »*



```
public class Car extends Vehicule {  
    private String couleur ;  
    private String marque ;  
    private float prix ;  
  
    public Car (String c, String m, float p) {  
        super(4)  
        couleur = c ;  
        marque = m ;  
        prix = p ;  
    }  
}
```

- Si la classe « B » hérite de la classe « A », on dira que « B » **extends** « A ».
- « B » hérite des méthodes de « A » et **peut les surcharger**

*Une instance de « Car »
peut utiliser les méthodes
de « Vehicule ».*

→ Polymorphisme*

```
public class UneAutreClasse {  
    ...  
  
    Car bmv = new Car (« vert », « bmv », 20000);  
    bmv.avancer();  
    ...  
}
```

Interfaces

Le principe

- Une classe peut implémenter une interface. Contrairement à une classe, une interface ne spécifie pas de comportement mais uniquement des définitions de méthodes.
- L'idée est d'imposer aux classes qui implémentent cette interface, une manière d'interagir avec l'extérieur.
- Il faut voir l'implémentation comme **un contrat** : la classe qui implémente une interface s'engage à surcharger toutes les méthodes définies dans cette interface.
- Une classe peut implémenter autant d'interfaces qu'elle le souhaite (...et hérité d'une classe).
- En Java, le mot clé est **implements**.

Attention!

Une interface ne peut contenir que des **variables constantes** ou **statiques** et des **entêtes de méthodes**

```
int UNE_VARIABLE_CONSTANTE = 1 ;
```


Interface

Interface et implémentation des méthodes


- Si la classe « B » implémente la classe « A, » on dira que « **B** » **implements** « **A** ».
- « B » s'engage surcharger toutes les méthodes de « A »

?

```
public interface Vehicule {  
  
    void rouler();  
    void freiner();  
}
```



```
public class Car implements Vehicule {  
    //Constructeurs  
    public Car(... , ..) { ...}  
  
    //Methodes  
    public void rouler() {  
        //Coder ici la manière dont l'auto roule  
    }  
    public void freiner() {  
        //Coder ici la manière dont l'auto freine  
    }  
  
    //Autres méthodes propres à Car.
```



```
public class Bike implements Vehicule {  
    //Constructeurs  
    public Bike(... , ..) { ...}  
  
    //Methodes  
    public void rouler() {  
        //Coder ici la manière dont le velo roule  
    }  
    public void freiner() {  
        //Coder ici la manière dont le velo freine  
    }  
  
    //Autres méthodes propres à Velo.
```

Polymorphisme

- **Concept** : Un objet n'est pas nécessairement manipulé selon la spécification de la classe dont il est une instance. Il peut être manipulé selon la spécification d'une **classe** dont il **hérite** (= qu'il étend) ou encore d'une **interface** qu'il **implémente**.
- On parle de **polymorphisme**.

Exemple :

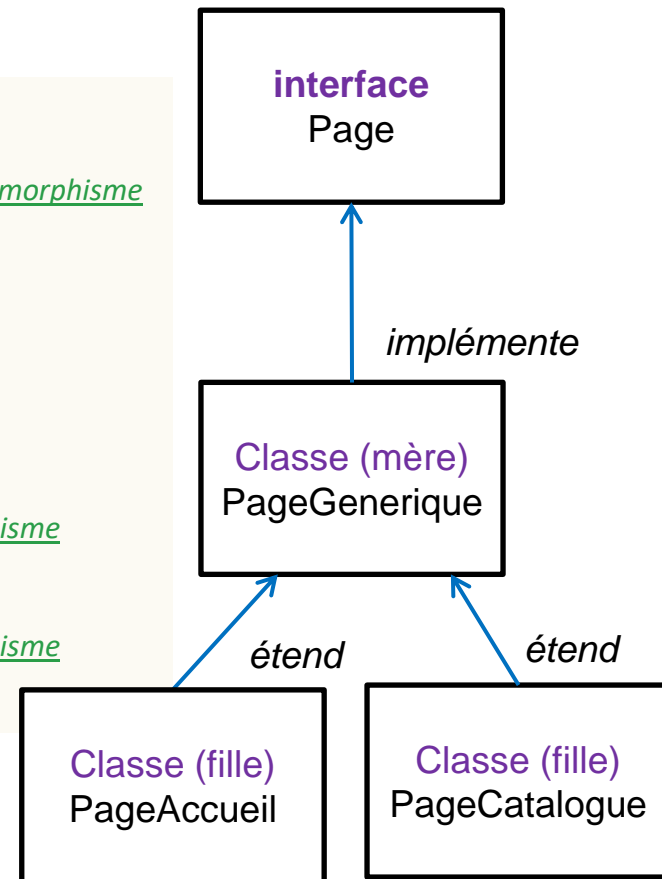
```
// création d'une instance de PageAccueil à partir du type PageGenerique -> Polymorphisme
PageGenerique page1 = new PageAccueil();

// création d'une instance de PageCatalogue à partir de son type -> OK
PageCatalogue page2 = new PageCatalogue();

// création d'une liste du type de l'interface Page
List<Page> listeDePages = new ArrayList<Page>();

// ajout de l'instance page1 (PageGenerique) dans la liste de pages -> Polymorphisme
listeDePages.add(page1);

// ajout de l'instance page2 (PageGenerique) dans la liste de pages -> Polymorphisme
listeDePages.add(page2);
```



Exercice 7

Utiliser l'héritage et l'interface

1. Créez une classe `Personne` de laquelle héritent les classes « `Eleve` », « `Enseignant` » et « `PersonnelAdministratif` »
2. Mutualisez les variables et méthodes communes aux trois classes filles dans la classe « `Personne` ». (Mettre en place un appel du constructeur de `Personne` avec la méthode `super()`).
3. Créez une interface « `EmployeEcole` » et définissez-y les méthodes communes aux classes « `Enseignant` » et « `PersonnelAdministratif` ». Implémentez l'interface à partir de ces dernières
4. Vérifiez que les tests de l'exercice 6 passent toujours en succès.
5. Modifiez vos tests de manière à prouver le principe de polymorphisme




Énumérations

Restreindre les valeurs possibles d'une variable

- Une énumération est un type de données particulier, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs. Ces valeurs sont des constantes nommées.
- **Exemple :** une énumération « Civilité » aura pour données : MADAME, MONSIEUR, MADEMOISELLE ;
- Une énumération se déclare comme une classe mais avec le mot-clé **enum** au lieu de **class**.



```
public enum Color {  
    BLEU,  
    VERT,  
    JAUNE,  
    ROUGE;  
}
```



```
public class Car {  
    private Color couleur ;  
    private String marque ;  
    private float prix ;  
  
    // public Car () {}  
  
    Public void setCouleur(Color c){  
        couleur = c ;  
    }  
  
    Public Color getCouleur(){  
        return couleur ;  
    }  
  
    ...  
}
```

```
public class UneAutreClasse {  
    ...  
    Car voiture1 = new Car();  
    voiture1.setCouleur(Color.BLEU);  
    ...  
}
```

Exercice 8

Créer une énumération

1. Créez une énumération appelée « NiveauClasse » avec les valeurs CP, CE1, CE2, CM1, CM2.
2. Dans la classe « Eleve », effectuez les modifications nécessaires pour que la variable d'instance `niveau_classe` ne puisse être valorisée qu'à partir de l'énumération « NiveauClasse »,
3. Ecrivez une méthode de test qui vous permet de vérifier la bonne attribution d'une classe à un élève.
4. Peut-on employer la même solution pour que l'âge d'un élève soit compris entre 6 et 10 ans ? Quelle solution pourrait-on envisager?



Décisions

- Mots-réservés : *if, else*

Une **condition** doit retourner « vrai » ou « faux »
→ Peut être toute méthode de retour **boolean**

- Structure :

if (condition){ <i>instructions</i> }	If (condition) { <i>instructions</i> } else { <i>instructions</i> }	If (condition 1) { <i>instructions</i> } else if (condition 2) { <i>instructions</i> }
-------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------

- Conditions :

Comparaison de types primitifs ou de référence d'objets	Comparaison de valeurs d'objets	Comparaisons d'objet avec <i>null</i>
<i>if</i> (<i>a == b</i>) <i>if</i> (<i>a != b</i>) <i>if</i> (<i>a <= b</i>) <i>if</i> (<i>a > b</i>)	<i>if</i> (a.equals(b)) <i>if</i> (!a.equals(b)) <i>if</i> ("".equals(b))	<i>if</i> (a == null) <i>if</i> (a != null)

Exercice 9

Utiliser les conditions et décisions

Dans les classes « Professeur » et « PersonnelAdministratif », créez une méthode `demandeAugmentation()` qui appelle la méthode de la classe « Ecole » permettant d'augmenter les salaires (exercice 6).

ATTENTION ! L'employé ne doit pas être augmenté si son nombre d'absence dans le mois dépasse 4.

Testez votre méthode....



Sélections

- Mots-réservés : **switch, case**
- Structure :

```
switch (variable de type primitif ou String){  
  
    case valeurDeLaVariable : instructions ; break;  
    case valeurDeLaVariable : instructions ; break;  
    case valeurDeLaVariable : instructions ; break;  
    ...  
    default : instructions;  
}
```

Besoin d'un
exemple avec
la voiture ?

```
Car voiture1 = new Car();  
switch (voiture1.getCouleur()){  
  
    case bleu : System.out.println (« belle voiture bleu ! ») ; break;  
    case vert : System.out.println (« belle voiture verte ! ») ; break;  
    case jaune : System.out.println (« belle voiture jaune ! ») ; break;  
    ...  
    default : System.out.println (« votre voiture n'a pas de couleur ? ») ;  
}
```

Exercice 10

Utiliser la gestion de cas

Dans la classe Ecole, créez une méthode capable d'assigner un niveau de classe à un élève selon son âge.

6 ans -> CP

7 ans -> CE1

8 ans -> CE2

...



For

Les boucles **for**

- Mots-réservés : **for**
- Structure :

```
for (initialisation variable; condition; instruction){  
    instructions  
}
```

Un exemple ?

```
for (int i=0; i<10; i++){  
    System.out.println( i );  
}
```



À chaque répétition de la boucle (il y en aura 10), la console affichera la valeur de *i*

While

Les boucles **while**

- Mots-réservés : **while** → (*tant que*)
- Structure :

```
while (condition){  
    instructions  
}
```

```
do {  
    instruction  
}  
while (condition);
```

L'*instruction* se répète tant que la *condition* est vrai (*true*)

Les collections listes et leur parcours

Créer une collection et lui affecter des valeurs

```
List<String> liste = new ArrayList<String>();  
liste.add("premier élément");  
liste.add("deuxième élément");
```

Parcourir la collection

```
for(String s : liste){  
    assertTrue(s.contains("élément"));  
}
```

Manipuler une liste

- `add(E element) : boolean`
- `addAll(List<E> elements) : boolean`
- `contains(Object o) : boolean`
- `isEmpty() : boolean`
- `size() : int`
- `get(int index) : Object o`

...

Exercice 11

Utiliser les collections et les boucles

Dans une classe de test, constituez une collection d'employés (4 enseignants et 3 membres du personnel administratif) et utilisez une boucle **for** pour que chacun d'eux se présentent.

La présentation doit s'afficher en console.



Tableaux

```
String tab[] = new String[10];  
tab[0] = "premier élément";  
tab[9] = "dernier élément";
```



Maps

- L'interface `Map<Key, Value>` permet de stocker des valeurs associées à des clés uniques.
- `containsKey(Object key) : boolean`
 - Retourne vrai si la clé existe, sinon retourne faux
- `containsValue(Object value) ; boolean`
 - Retourne vrai si la valeur existe, sinon retourne faux
- `put(K key, V value) : V`
 - Ajoute une correspondance clé/valeur
- `get(Object key) : V`
 - Permet de récupérer la valeur associée à une clé
- `keySet() : Set<K>`
 - Permet de récupérer l'ensemble des clés

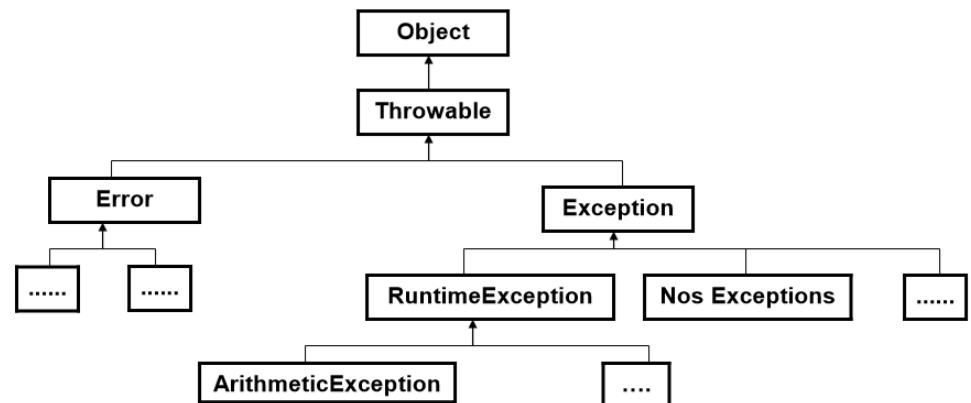
<https://examples.javacodegeeks.com/java-map-example/>

Exceptions

Concepts

- Une **exception** représente une **erreur**.
- Une exception est un signal qui se déclenche en cas de problème.
- Il est possible de **lancer** une exception pour signaler une erreur.
- Lancer une exception si elle n'est pas gérée implique l'interruption du programme
- Pour éviter l'arrêt du programme on peut **gérer** les erreurs
- La gestion des exceptions se décompose en deux phases :
 - La levée d'exceptions,
 - Le traitement d'exceptions

Les exceptions sont
des classes Java



Exceptions

Créer et lever des exceptions : **throws**

- Il existe de nombreuses classes d'exception, dont beaucoup sont levées implicitement par la machine virtuelle.
- Il est néanmoins possible de créer des classes d'exception spécifiques que l'on entend lever sous certaines conditions
- Dans une méthode, on prévoit la levée de l'exception par les mots-clés **throws** et **throw**

X

```
public class SaisieErroneeEx extends Exception {  
  
    public SaisieErroneeEx(String s) {  
        super(s);  
    }  
  
}
```



```
public class voiture {  
    ...  
    public setMarque (String marque) throws SaisieErroneeEx {  
        if (marque.equals("")) {  
            SaisieErroneeEx("Saisie erronee : chaine vide");  
        }  
        ...  
    }  
}
```

Exceptions

Gérer les exception : **try** , **catch**

Le traitement des exceptions se fait à l'aide de la séquence d'instruction **try...catch...finally**.

```
try{  
    // instructions susceptibles de lever des exceptions  
} catch(Exception e){  
    // instructions si une exception est levée  
} finally {  
    // Sert à définir un bloc de code à exécuter dans tous les cas  
}
```

Les tests non-passant

@Test (expected=Exception.class, timeout=1000)

Permet de déclarer que le déclenchement d'une exception est une condition de succès du test, de fixer un timeout en millisecondes

Exercice 12

Utiliser les exceptions

A l'instanciation d'un élève, son âge doit être compris entre 6 et 10 ans, sinon une erreur est envoyée.

Ecrivez votre test avant de développer la méthode. Votre test doit être en échec jusqu'à ce que votre méthode soit correctement implantée...

C'est le TDD, le *test driven development*. BRAVO



Le logging

Le principe

Jusque là, nous avons utilisé la méthode **System.out.println** pour stocker des informations ou des erreurs au cours de nos exécutions.

En développement, c'est une TRES mauvaise pratique !

Pour cela, il existe des API de logging

Il est fortement recommandé d'utiliser une API de logging plutôt que d'utiliser la méthode `System.out.println()` pour plusieurs raisons :

- Permet de gérer le **niveau de gravité** des messages pris en compte (INFO, WARN, ERROR, DEBUG...)
- Permet un contrôle sur le **format des messages** en proposant un format standard pouvant inclure des données telles que la date/heure, la classe ...
- Une API de logging permet de **gérer différentes cibles de stockage** des messages (ex : fichiers externes)

Le logging

Le principe

1. Chaque classe doit instancier un logger de manière à tracer la classe émettrice des logs
2. On ne doit plus voir de **System.out.println** trainer dans les classes
3. Les niveaux de gravité des logs doivent être en adéquation avec le message
4. Chaque message doit contenir *a minima* la date/heure d'émission



Le logging

Mise en place

**Nous utiliserons l'API de logging SL4J
(qui implémente log4j)**

Trois dépendances à ajouter dans le POM.xml :

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.12.1</version>
</dependency>

<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.12.1</version>
</dependency>

<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j-impl</artifactId>
  <version>2.12.1</version>
</dependency>
```



Le logging

Mise en place

1. Chaque classe doit instancier un logger de manière à tracer la classe émettrice des logs
2. On ne doit plus voir de **System.out.println** trainer dans les classes

```
public class Voiture {  
  
    private String couleur;  
    private String marque;  
    private double prix;  
  
    public void setCouleur(String couleur) {  
        this.couleur = couleur;  
        System.out.println("La voiture est "+couleur);  
    }  
}
```

```
public class Voiture {  
  
    private String couleur;  
    private String marque;  
    private double prix;  
  
    static Logger logger = LoggerFactory.getLogger(Voiture.class);  
  
    public void setCouleur(String couleur) {  
        this.couleur = couleur;  
        logger.info("La voiture est "+ couleur);  
    }  
}
```

Le logging

Mise en place

3. Les niveaux de gravité des logs doivent être en adéquation avec le message

```
public class Voiture {  
  
    private String couleur;  
    private String marque;  
    private double prix;  
  
    static Logger logger= LoggerFactory.getLogger(Voiture.class);  
  
    public void setCouleur(String couleur) throws Exception {  
        if(couleur.isEmpty()){  
            logger.error("saisie eronée");  
            throw new Exception();  
        }  
        else if (couleur.equals("bleu") | couleur.equals("vert") | couleur.equals("rouge")) {  
            this.couleur = couleur;  
            logger.info("la voiture est de couleur "+ couleur);  
        }  
        else {  
            logger.warn("la couleur demandée n'est pas disponible");  
        }  
    }  
}
```

5 niveaux de criticité

TRACE

DEBUG

INFO

WARN

ERROR

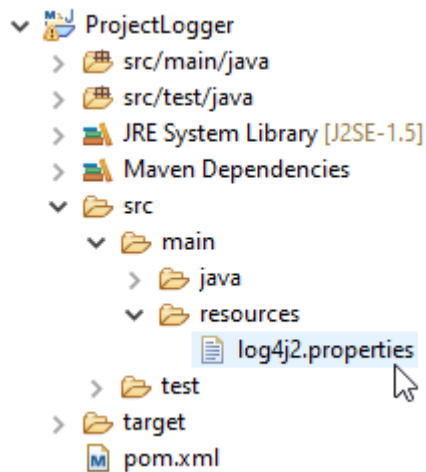
Question bonus : comment pourrait-on rendre cette gestion d'erreur plus efficace sur le choix de la couleur ?

Le logging

Mise en place

4. Chaque message doit contenir *a minima* la date/heure d'émission

- Créer un fichier log4j2.properties dans src/main/resources
- Configurer le pattern et ajouter une variable %d{...}
- Configurer le niveau de log de l'exécution



```
appender.console.type = Console
appender.console.name = STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L - %m%n

rootLogger.level = info
rootLogger.appenderRefs = stdout
rootLogger.appenderRef.stdout.ref = STDOUT
```

Exercice 13

Utiliser le Logger

Reprendre le test de l'exercice 9 et configurer le Logging de manière à ce que la console n'affiche que les logs SLF4J



Annotations JUnit

- `@Test`
 - Permet d'identifier une méthode en tant que test
- `@Test(expected=Exception.class, timeout=1000)`
 - Permet de déclarer que le déclenchement d'une exception est une condition de succès du test, de fixer un timeout en millisecondes
- `@Before`
 - Permet d'identifier une méthode comme traitement lancé avant chaque test de la classe
- `@After`
 - Permet d'identifier une méthode comme traitement lancé après chaque test de la classe
- `@BeforeClass`
 - Permet d'identifier une méthode publique et **statique** comme traitement lancé une fois avant les tests de la classe
- `@AfterClass`
 - Permet d'identifier une méthode publique et **statique** comme traitement lancé une fois après les tests de la classe
- `@Ignore`
 - Permet d'ignorer un test

Paramètres en JUnit 4

`@RunWith(Parameterized.class)`

```
public class CalculatriceTest {
```

```
    int operande1, operande2, resultat;
```

```
    public CalculatriceTest(int operande1, int operande2, int resultat) {
```

```
        super();
```

```
        this.operande1 = operande1;
```

```
        this.operande2 = operande2;
```

```
        this.resultat = resultat;
```

```
    }
```

`@Parameters`

```
    public static Collection values() {
```

```
        return Arrays.asList(new Object[][] {
```

```
            {1, 1, 2 }, // 1+1=2
```

```
            {2, 2, 4 }, // 2+2=4
```

```
            {2, 2, 5 } // 2+2=5 ???
```

```
        });
```

```
    }
```

Suite de tests en JUnit 4

```
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)
```

```
@Suite.SuiteClasses({
```

```
    CalculatriceTest.class,
```

```
    CalculatriceTest2.class })
```

```
public class CalculatriceTestSuite { }
```



Mots réservés – Types primitifs

abstract	assert	boolean	break	byte	case	catch
char	class	<i>const</i>	continue	default	do	double
else	enum	extends	false	final	finally	float
for	<i>goto</i>	if	implements	import	instanceof	int
interface	long	native	new	null	package	private
protected	public	return	short	static	strictfp	super
switch	synchronized	this	throw	throws	transient	true
try	void	volatile	while			



Mots réservés – Structures de contrôle

abstract	assert	boolean	break	byte	case	catch
char	class	<i>const</i>	continue	default	do	double
else	enum	extends	false	final	finally	float
for	<i>goto</i>	if	implements	import	instanceof	int
interface	long	native	new	null	package	private
protected	public	return	short	static	strictfp	super
switch	synchronized	this	throw	throws	transient	true
try	void	volatile	while			



Mots réservés – Gestion des exceptions

abstract	assert	boolean	break	byte	case	catch
char	class	<i>const</i>	continue	default	do	double
else	enum	extends	false	final	finally	float
for	<i>goto</i>	if	implements	import	instanceof	int
interface	long	native	new	null	package	private
protected	public	return	short	static	strictfp	super
switch	synchronized	this	throw	throws	transient	true
try	void	volatile	while			

