Types Abstraits & Bases de la POO
V - Bases de la Programmation Objet : Java

- - - - - - - - - -
1 – Les bases
2 – Les classes
3 – Les instances
4 – Concepts de la programmation objet
- - - - - - - - - -

ENSMA A3-S5 - période A

2021-2022

M. Richard
richardm@ensma.fr

Sommaire

I - Les bases

- Généralités
- Syntaxe
- Les types
- Structures de contrôle
- Documentation : `javadoc`

II - Les classes

- Définition
- Portée des entités
- Les attributs
- Les méthodes
- Les méthodes statiques

III - Les instances

- Instanciation
- Recopie
- Comparaison
- Valeur "`null`" & Objet "`this`"
- Manipulation des instances
- Constructeurs
- Destructeurs

IV - Concepts de la programmation objet

- Surcharge
- Redéfinition
- Encapsulation
- Héritage
- Polymorphisme
- Abstraction
- Interface

I - Les bases

1 – Généralités . . .
2 – Syntaxe
3 – Les types
4 – Structures de contrôle
5 – Documentation : `javadoc`

↪ Naissance de Java

Origine :

- Créé par SUN
- Cible : les systèmes embarqués (véhicules, électroménager, etc..) utilisant des langages dédiés et incompatibles entre eux

Dates-clés :

- 1990 : Introduction du langage 'OAK' par James Goslin
- 1993 : Montée en puissance du Web grâce à Mosaic (l'idée d'adapter Java au Web fait son chemin)
- 1995 : Java 1.0 officiellement lancé en mai 95
- 1996 : NetscapeTM Navigator 2 incorpore une machine virtuelle Java 1.0 en version 'bêta'
- 1997 : Un premier pas vers une version industrielle Java 1.1
- 1999 : Version industrielle de Java (version 1.2)

↪ Caractéristiques

- Un langage de programmation orienté objet (POO)
- Une machine virtuelle (JVM)
  - joue le rôle d'interpréteur
- Bibliothèques de classes standards = API (Application Programming Interface)
  - environ 30000 classes dans Java 1.5
- Ensembles d'outils
  - java, javac, jdb, javadoc, jar, ...

Définition de Java donnée par SUN :

- Simple, sûr, orienté objet,
- portable, réparti,
- performant, interprété,
- multitâches, robuste,

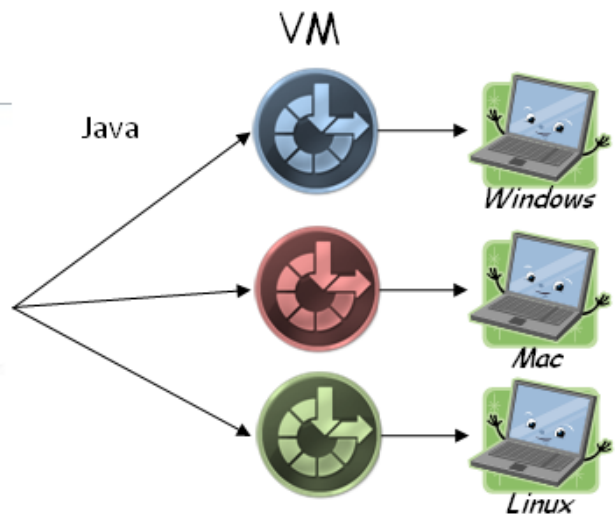
↪ Langage compilé/interprété

- Compilation en pseudo code, appelé byte-code, puis exécution par un interpréteur Java (la Java Virtual Machine (JVM))
  - multi-plateformes :
    - Navigateur Web, Station de Travail, Network Computeur
    - WebPhones
    - Cartes à puces
    - ...

```java
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        for(int i=0;i<10;i++)
        {
            System.out.println("Prase "+i+" : Coucou Tout le monde !!!");
        }
    }
}
```

Javac

```
 0 iconst_0
 1 istore_1
 2 goto 35 (+33)
 5 getstatic #16 <java/lang/System.out>
 8 new #22 <java/lang/StringBuilder>
11 dup
12 ldc #24 <Prase >
14 invokespecial #26 <java/lang/StringBuilder.<init>>
17 iload_1
18 invokevirtual #29 <java/lang/StringBuilder.append>
21 ldc #33 < : Coucou Tout le monde !!!>
23 invokevirtual #35 <java/lang/StringBuilder.append>
26 invokevirtual #38 <java/lang/StringBuilder.toString>
29 invokevirtual #42 <java/io/PrintStream.println>
32 iinc 1 by 1
35 iload_1
36 bipush 10
38 if_icmplt 5 (-33)
41 return
```



↪ Les versions de Java 1/3

Les sigles :

- JDK : Java Developpement Kit
- JRE : Java Runtime Edition
  - doit être installé pour exécuter des applications Java
- Version 1.2 = Java 2
- JDK 1.2 → J2SDK
- JRE 1.2 → J2RE

Trois éditions du JDK :

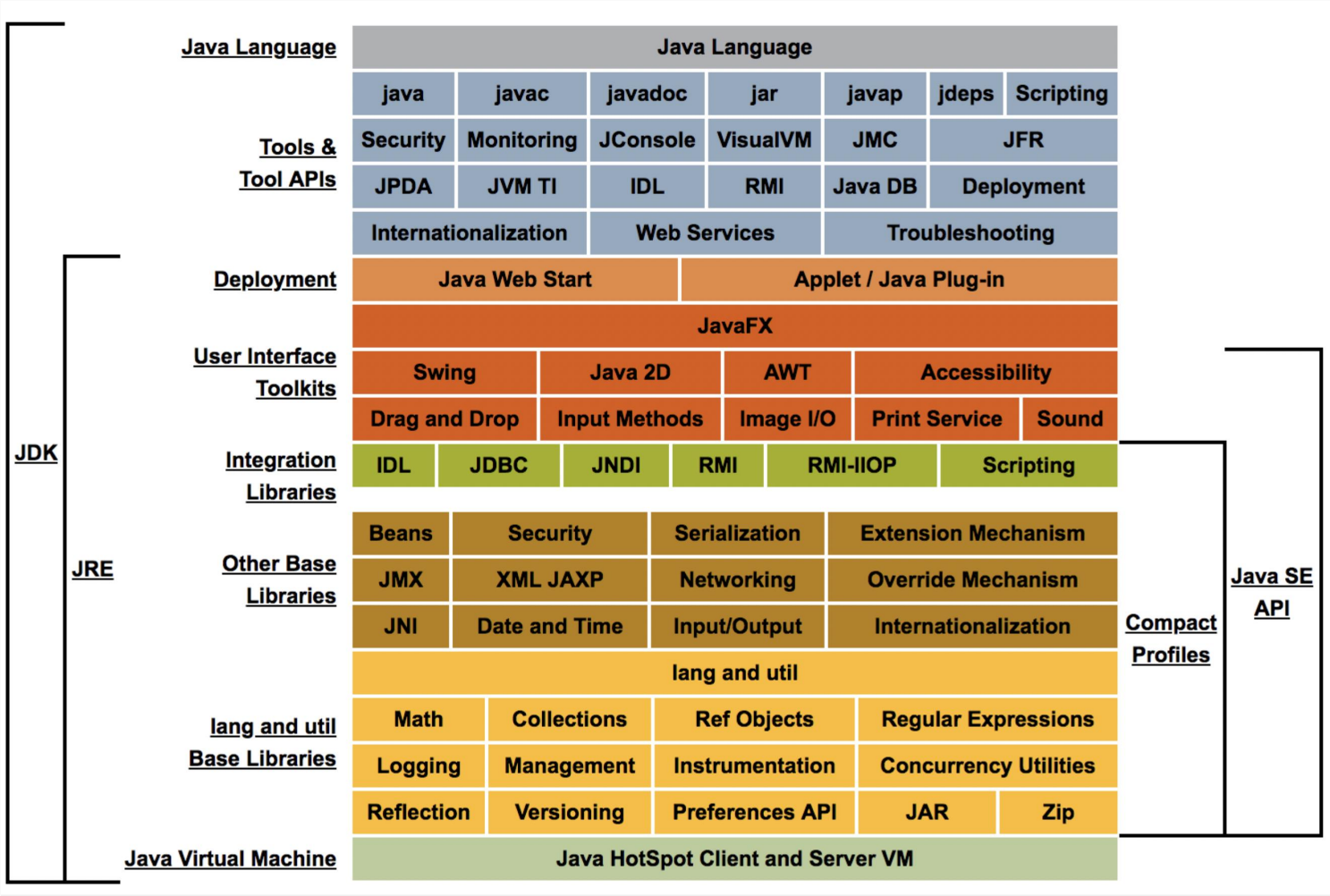
- J2ME : Java 2 Mobile Edition
  - environnement optimisé pour les dispositifs mobiles
- J2SE : Java 2 Standard Edition
  - fournit tous les outils nécessaires au développement, à l'exécution et au déploiement d'applications et d'applets
- J2EE : Java 2 Entreprise Edition
  - Dédiée au développement d'application distribuées et articulées autour du Web
- JDK fourni sous différentes plate-formes :
  - Windows, Solaris, Linux

↪ Les versions de Java 2/3

- JDK 1.0
  - première version lancée officiellement en mai 1995
- JDK 1.1
  - annoncée officiellement en mars 1997
- JDK 1.2
  - Cette version est lancée fin 1998 ; marque le début de l'essor de Java
- JDK 1.5 (Nom de code Tiger)
  - début 2005
    - simplifier les développements en Java et répondre à C# ...
  - Nouvelles fonctionnalités :
    - Autoboxing/Unboxing
    - Arguments variables
    - Generics
    - boucles pour les collections
    - Énumérations
- JDK 1.11, JSE11 → 09/2018
  - LTS

↪ Les versions de Java 3/3

- Version utilisée : OpenJDK 11



↪ Les outils

IDE : Integrated Developement Environment

- Free :
  - Eclipse
  - NetBeans
  - JCreator LE
  - Emacs + JDE
  - BlueJ
  - ...

Autres :

- Maven : outils de configuration/construction de projets
- git : outils de gestion de versions (pour les sources d'un projet)

↪ Structure

- Java est sensible à la casse
- Les blocs de code sont encadrés par des accolades.
- Chaque instruction se termine par un caractère ' ;'
- Une instruction peut tenir sur plusieurs lignes
- L'indentation est ignorée par le compilateur

↪ Identificateurs

- Chaque objet, classe, programme ou variable est associer à un nom : l'identificateur
- Composé de tous les caractères alphanumériques et des caractères _ et $
- Le premier caractère doit être une lettre, le caractère de soulignement ou le signe dollars
- Ne doit pas être un mot réservé du langage (i.e. mot clé)

↪ Mots Clés

- `abstract, boolean, break, byte,case,catch, char,class, continue,default, do, double, else, extends,final, finally, float, for, if, implements, import, instanceof, int, interface, long, native, new, null,package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while`

↪ Convention de nommage

- noms de classe commencent par une majuscule
- mots contenus dans un identificateurs à l'intérieur d'une classe commencent par une majuscule
  - `MaClasse`
  - `maMethode( ... )`
  - `maVariable`
- constantes en majuscules et mots des constantes séparés par _
  - `MA_CONSTANTE`
- si variable locale : tout en minuscule ou préfixé par _
  - `mavariable` ou `_maVariable`

↪ Commentaires

- Ils ne sont pas pris en compte par le compilateur
  - mais certains peuvent être interprétés pour générer la documentation d'API
- Ils ne se terminent pas par un ;
- trois types de commentaire :

Type de commentaires	Exemple
Commentaire abrégé	`//commentaire sur une seule ligne`
Commentaire multi-ligne	`/* commentaire ligne 1` `commentaire ligne 2 */`
Commentaire de documentation	`/** interprété par javadoc */`

↪ Les types primitifs 1/5

- Ne sont pas des Classes :
  - Emplacement mémoire réservé en mémoire dès sa déclaration ...

  Chacun des types simples possède un alter-ego objet disposant de méthodes de conversion (à voir dans la partie Classes et Objets)
  - `int` → `Integer`
  - `float` → `Float`
  - ...
- L'autoboxing :
  - permet de convertir de manière transparente les types primitifs en références
- Types élémentaires indépendants de l'architecture :
  - contrairement au C/C++
  - comme le C#
- Types élémentaires et valeurs :

Type	Taille	Valeurs
boolean	1	true/false
byte	8	-2^7 à $+2^7$
char	16	0 à 65535
short	16	-2^{15} à $+2^{15}-1$
int	32	-2^{31} à $+2^{31}-1$
long	64	-2^{63} à $+2^{63}-1$
float	32	1.40239846e-45 à 3.40282347e38
Long	64	4.9406564584124654e-324 à 1.79769313486231570e308

↪ Les types primitifs 2/5

Nombres entiers :

- Les types `byte`, `short`, `int` et `long` peuvent être codés en décimal, hexadécimal ou octal.
  - Pour un nombre hexadécimal, il suffit de préfixer sa valeur par 0x.
  - Pour un nombre octal, le nombre doit commencer par un zéro.
- Le suffixe `l` ou `L` permet de spécifier que c'est un entier long.

Nombres décimaux :

- pour être reconnus comme flottants, ils doivent posséder soit :
  - un point,
  - un exposant
  - l'un des suffixes `f`, `F`, `d`, `D`.
- Il est possible de préciser des nombres qui n'ont pas de partie entière ou décimale.
  - Exemple :

```
float pi = 3.141f;
double v = 3d;
float f = +.1f , d = 1e10f;
```

↪ Les types primitifs 3/5

Caractères :

- Un caractère doit être entouré par des apostrophes.
- Une valeur de type char peut être considérée comme un entier non négatif de 0 à 65535.
- la conversion implicite par affectation n'est pas possible.

Initialisation :

- attribut : initialisé avec une valeur par défaut en accord avec son type au moment de sa création.
- attention, pas d'initialisation pour les variables locales ...

Type	Valeur par défaut
Boolean	False
Byte, short, int, long	0
Float double	0.0
Char	\u000

↪ Les types primitifs 4/5

Cast : conversion entre numérique

- attention à la perte d'information ...

```
int entier = 5;
float flottant = (float) entier;
```

Cast : conversion chaîne/numérique

- pas d'opérateur → utilisation de méthodes fournit dans les classes correspondant aux types élémentaires
  - Classe String, Integer, Long, Float et Double
  - Pour y accéder, il faut les instancier puisque se sont des objets.
  - Exemple :

```
String montexte;
montexte = new String("test");
```

  - `montexte` permet d'accéder aux méthodes de la classe `java.lang.String` ; c'est un objet

↪ Les types primitifs 5/5

Entier int en chaîne de caractère String

- Exemple :

```
int i = 10;
String montexte = new String();
montexte =montexte.valueOf(i);
```

  - `valueOf` également définie pour les types boolean, long, float, double et char

Chaîne de caractères String en entier int

- Exemple :

```
String montexte = new String("10");
Integer nomnombre=new Integer(montexte);
//convertion d'Integer en int
int i = monnombre.intValue();
```

Entier int en entier long

- Exemple :

```
int i=10;
Integer monnombre=new Integer(i);
long j=monnombre.longValue();
```

↪ Types primitifs & opérateurs 1/7

Caractère : `char touche = '%';`

Chaîne : `String texte = "bonjour" ;`

- Les variables de type `String` sont des objets
- Les chaînes ne sont pas des tableaux :
  - il faut utiliser les méthodes de la classes `String` pour effectuer des manipulations.
- Java ne fonctionne pas avec le jeu de caractères ASCII ou ANSI, mais avec Unicode (Universal Code).
  - Unicode code un caractère sur 2 octets
  - Les caractères 0 à 255 correspondent au jeu de caractères ASCII

Caractères spéciaux	Affichage	Caractères spéciaux	Affichage
\'	Apostrophe	\f	saut de page (form feed)
\ »	Guillemet	\n	saut de ligne (newline)
\\	anti slash	\0ddd	caractère ASCII ddd (octal)
\t	Tabulation	\xdd	caractère ASCII dd (hexadécimal)
\b	retour arrière	\udddd	caractère Unicode dddd (hexadécimal)
\r	retour chariot		

↪ Types primitifs & opérateurs 2/7

Pour les entiers, Java met en œuvre un mécanisme de conversion implicite vers le type int appelé promotion entière.

- pour renforcer la sécurité du code.
- Exemple :

```
short x= 5 , y = 15;
x = x + y ; //erreur à la compilation

Incompatible type for =. Explicit cast needed to
convert int to short.
x = x + y ;
^
1 error

=> x = (short) ( x + y );
```

L'opération doit être entre parenthèse pour définir la portée :

- le cast à une priorité plus forte que les autres opérateurs

La division par zéro pour les types entiers lève l'exception `ArithmeticException`

↪ Types primitifs & opérateurs 3/7

Valeurs float ou double : la division par zéro ne produit pas d'exception mais le résultat est indiqué par une valeur spéciale qui peut prendre trois états :

- indéfini, infini positif ($+\infty$) ou infini négatif ($-\infty$)

```
Float.NaN ou Double.NaN (not a number)
Float.POSITIVE_INFINITY ou Double.POSITIVE_INFINITY
Float.NEGATIVE_INFINITY ou Double.NEGATIVE_INFINITY
```

Conformément à la norme IEEE754, ces valeurs spéciales représentent le résultat d'une expression invalide NaN, une valeur supérieure au plafond du type pour infini positif ou négatif.

X	Y	X/Y	X%Y
valeur finie	0	+∞	NaN
valeur finie	+/-∞	0	X
0	0	NaN	NaN
+/-∞	Valeur finie	+/-∞	NaN
+/-∞	+/-∞	NaN	NaN

↪ Types primitifs & opérateurs 4/7

Opérateur d'affectation = :

- expression de la forme variable = expression.
- associatif de droite à gauche
- renvoie la valeur affectée

```
x = y = z = 0;
```

Opérateurs de simplification :

- permettant de simplifier l'écriture d'une opération d'affectation associée à un opérateur mathématique :

Opérateur	Exemple	Signification
=	a=10	a=10
+=	a+=10	a=a+10
-=	a-=10	a=a-10
*=	a*=10	a=a*10
/=	a/=10	a=a/10
%=	a%=10	Reste de la division
^=	a^=10	a=a^10
<<=	a<<=10	a=a<<10 complété par des 0 à droite
>>=	a>>=10	a=a>>10 complété par des 0 à gauche
>>>=	a>>>10	Décalage à gauche non signée

↪ Types primitifs & opérateurs 5/7

Opérateurs de comparaison :

Opérateur	Exemple	Signification
>	a>10	Strictement supérieur
<	a<10	Strictement inférieur
>=	a>=10	Supérieur ou égale
<=	a<=10	Inférieur ou égale
==	a==10	Égalité
!=	a!=10	Différent de
&	a&b	ET binaire
^	a^b	OU exclusif binaire
|	a|b	OU binaire
&&	a&&b	ET logique : l'évaluation de l'expression cesse dès qu'elle devient fausse
||	a||b	OU logique : l'évaluation de l'expression cesse dès qu'elle devient vraie
? :	a ? b : c	opérateur conditionnel : renvoie la valeur b ou c selon l'évaluation de l'expression a

↪ Types primitifs & opérateurs 6/7

Incréments/Décréments (comme en C)

- `n++, ++n, n--, --n`

Opérateur placé avant la variable : préfixé

- modification de la valeur immédiate

Opérateur placé après la variable : postfixé

- modification n'a lieu qu'à l'issu de l'exécution de la ligne d'instruction

Exemple :

```
// est équivalent à System.out.println(x); x = x + 1;
System.out.println(x++);

// est équivalent à x = x + 1; System.out.println(x);
System.out.println(++x);
```

↪ Types primitifs & opérateurs 7/7

Priorité des opérateurs :

Parenthèses	()
incrémentation	++, --
multiplication, division, et modulo	*, /, %
d'addition et soustraction	+, -
décalage	<<, >>
comparaison	<, >, <=, >=
Égalité	==, !=
OU exclusif	^
ET	&
OU	|
ET logique	&&
OU logique	||
Assignement	=, +=, -=

↪ Itération 1/2

- Tant Que Vrai ... Faire

```
while ( boolean )
{
  ... // code a exécuter dans la boucle
}
```

- Faire ... Tant Que Vrai

```
do
{
  ... // code a exécuter dans la boucle
} while ( boolean )
```

  - il y aura toujours au moins une exécution ...

↪ Itération 2/2

- Pour ... Faire

```
for ( initialisation; condition; modification) {
  ... // code a exécuter dans la boucle
}
```

- Initialisation, condition et modification de l'index sont optionnels :

```
for ( ; ; ) { ... } // boucle infinie
```

- Déclaration de variable servant d'index autorisée dans l'initialisation

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

  - variable locale à la boucle
  - pas de déclaration à l'extérieur

- plusieurs traitements dans l'initialisation et la modification de la boucle

```
for (i = 0 , j = 0 ; i * j < 1000;i++ , j+= 2) { ....}
```

  - les traitements sont séparés par une virgule

↪ Conditionnelle

- Si ... Alors ... Sinon ...

```
if (boolean) {
  ...
}else if (boolean) {
  ...
}else {
  ...
}
```

- Switch ... Case ...

```
switch (expression) {
  case constante1 :
  instr11;
  instr12;
  break;
  case constante2 :
  ...
  default :
  ...
}
```

  - si break est omis, le "case" suivant est exécuté
  - Utilisable qu'avec des types primitifs d'une taille maximum de 32 bits
    - byte, short, int, char

↪ Débranchement

- `break`
  - permet de quitter immédiatement une boucle ou un branchement. Utilisable dans tous les contrôles de flot
- `continue`
  - s'utilise dans une boucle pour passer directement à l'itération suivante
- non recommandé ... limiter l'utilisation de ces deux instructions

↪ Principe

- Java propose un utilitaire javadoc permettant la création de documentation d'API ou technique
- nécessite l'ajout de commentaires dans les fichiers de code

```
/**
* Informations pour la documentation technique ...
*/
public class MaClasse {
  ...
}
```

- javadoc extrait les informations des commentaires placés au dessus des éléments suivants :
  - package (fichier package-info)
  - classe, enum, interface,
  - méthodes publiques et protégées,
  - attributs publics et protégés
- génére un ensemble de fichier Html reliés entre eux :
  - visualisable et navigable dans un navigateur
  - même style que la documetation techique de l'API Java

↪ Annotations

Commentaires de classe/Package :

- placé entre les instructions d'import et le début de la définition de la classe
- utilisation des annotations générales si nécessaire

Commentaires de méthodes :

- précède immédiatement la méthode décrite
- utilisation des balises générales
- utilisation de balises spécifiques pour décrire la signature de la méthode :
  - `@param` : description d'un paramètre
  - `@return` : description de la valeur de retour si nécesssaire
  - `@throws` : descritpion de(s) exception(s) possiblement générée(s) par la méthode

Commentaires d'attributs :

- uniquement pour les attributs publics
- pas d'utilisation d'annotation particulière

↪ Annotations

Liste des balises générales les plus courantes :

Annotation	Utilisation
`@author` nom	auteur
`@version` numéro de version	numéro de version de l'entité commentée
`@deprecated` texte	entitée dépréciée et ne devant plus être utilisée
`@since` version	indique le numéro de version dans laqelle a été intoduite l'entité
`@see` reférence	permet l'ajout d'un lien hypertexte vers un élément

II - Les classes

1 – Définition
2 – Portée des entités
3 – Attributs
4 – Méthodes
5 – Méthodes statiques

Une classe est formée de :

- données : attributs
- procédures : méthodes

Une classe est un modèle pour des objets ayant :

- même structure,
- même comportement

Les objets sont des représentations dynamiques (i.e. instanciation) du modèle défini par la classe

- une classe permet d'instancier plusieurs objets
- un objet est instance d'une et une seule classe
  - pas d'héritage multiple

↪ Signature

- `modificateurs nom_de_classe [extends classe_mere] [implements interface] { ...}`

```
ClassModifiers class ClassName [extends SuperClass]
[implemenents Interfaces]
{
  // insérer ici les champs et les méthodes
}
```

Modificateur	Rôle
abstract	Une classe déclarée abstraite ne peut être instanciée. Il est donc nécessaire de définir une classe concrète héritant de la classe abstraite et implémentant donc l'ensemble des méthodes. Si une classe possède au moins une méthode abstraite, alors la classe est forcément abstraite.
final	La classe ne peut être héritée car ce modificateur interdit la redéfinition de la classe par héritage.
private	La classe n'est accessible qu'à partir du fichier dans laquelle elle est définie.
public	la classe est accessible à toutes les autres classes de son package. Elle devient accessible à une classe d'un autre package si son propre package est importée dans cette classe.

↪ Signature

- abstract et final ainsi que public et private sont mutuellement exclusifs.
- extends
  - ce mot clé permet de spécifier une superclasse éventuelle :
  - ce mot clé permet de préciser la classe mère dans une relation d'héritage.
- implements
  - ce mot clé permet de spécifier une ou des interfaces que la classe implémente.
- L'ordre des méthodes dans une classe n'a pas d'importance
  - Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée dans A.
- exemple :

```
1  public class UnExemple {
2    //Attributs
3
4    //Methodes
5  }
```

↪ Définition

Les opérateurs de portée réglementent l'accès aux aux méthodes et aux attributs des instances des classes (i.e. des objets). On entend par entité d'une instance de classe :

- ses données, i.e. ses attributs ;
- son comportement, i.e. ses méthodes.

D'une manière générale, 3 modificateurs permettent de définir la visibilité des entités :

- `public`, `private` et `protected` définissent des niveaux de protection différents

Modificateur	Rôle
`public`	L'entité est visible et accessible à toutes autres instances. On remarque ici que dans le cas d'un attribut, celui-ci ne devrait être qualifié de publique que dans des situations très précises.
`protected`	L'entité est visible et accessible à toutes instances de la classe elle-même et de toutes les instances des classes de sa hiérarchie d'héritage
`private`	Protection la plus forte. L'entité n'est accessible qu'aux instances de la classe elle-même.

↪ Définition

Ce sont des variables qui peuvent être

- des variables d'instances,
- des variables de classes
  - constantes ou non

↪ Déclaration

- `modificateurs type nom_attribut ;`
- modificateurs :
  - visibilité (Cf. précédemment) + static et final
- type :
  - type primitif ou classe

↪ Attributs d'instance

- simple déclaration
- chaque instance possède sa propre occurrence de l'attribut

↪ Attributs de classe

- définition précédée du mot clé `static`
- les instances de la classe partagent une unique occurrence de l'attribut

↪ Constantes

- sont logiquement des attributs de classe (puisque leur état n'évloue pas)
  - `static`
- utilisation du mot clé `final`
  - pas de modification de leur valeur

Rappel :

- dans la majorité des situations, pas d'attributs `public`

Exemple :

```
1   public class ClasseComplexe {
2
3     //Attribut de classe
4     private static int nbInstComplexe;
5
6     //Attribut d'instance
7     private float reComplexe;
8     private float imComplexe;
9
10    //Constante
11    private static final float PI = 3.14f;
12  }
```

↪ Définition

Envoi de message :

- demande à un objet d'effectuer une opération

Un message est composé de trois parties

- une référence
  - permet de désigner l'objet destinataire
- un nom de méthode
  - permet de préciser l'opération à effectuer
- une liste de paramètres éventuels
  - lorsque la méthode le nécessite

Rappel :

- Pour accéder aux méthodes d'un objet, on utilise la notation pointée
  - `monObjet.maMethode()`

↪ Signature

```
modificateurs type_retourné nom_méthode ( arg1, ... ) {
  // définition des variables locales et du bloc d'instructions
}
```

Le type retourné

- élémentaire
- une classe
- mot clé `void`
  - Si la méthode ne retourne rien (i.e. procédure)

Pas de valeurs par défaut dans les paramètres.

Arguments passés par valeur :

- la méthode fait une copie de la variable qui lui est locale
- si objet :
  - reçoit une référence qui désigne son emplacement mémoire d'origine
    - copie de la variable .
  - modification de l'objet possible via ses méthodes
  - impossible de remplacer la référence contenue dans la variable passée en paramètre
    - ce changement n'aura lieu que localement à la méthode

↪ Modificateurs

Modificateur	Rôle
`public`	La méthode est accessible à toutes autres instances.
`protected`	La méthode est accessible à toutes instances de la classe elle-même et de toutes les instances des classes de sa hiérarchie d'héritage.
`private`	La méthode n'est accessible qu'aux méthodes de la classe elle-même.
`final`	La méthode ne pourra être redéfinie au sein d'une classe héritant de celle dans laquelle elle est définie.
`static`	La méthode est accessible via la classe, et non, comme dans les autres cas, via une instance de la classe.

Type de la valeur de retour :

- doit être transmise par l'instruction `return`
- indique la valeur que prend la méthode et termine celle-ci
- toutes les instructions qui suivent `return` sont donc ignorées.
- Autre utilisation : Il est possible d'inclure une instruction `return` dans une méthode de type `void`
  - cela permet de quitter la méthode.

↪ Signature

```
[modificateur] static <typeRetour> nomMethode( [liste de paramètres] ){
  //corps de la méthode
}
```

Attention :

- Comme pour les attributs, une méthodes taguée `static` est commune à l'ensemble des instances ; c'est une méthode de classe
  - on y accède via la classe et non l'instance :
  - passage par valeur

↪ La méthode `main`

- méthode statique particulière
  - toujours la même signature
  - point d'entrée de l'application (doit être présente une unique fois dans l'ensemble des classes)

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

III - Les instances

1 – Instanciation
2 – Recopie
3 – Comparaison
4 – Valeur null & objet this
5 – Manipulation des instances
6 – Constructeurs
7 – Destructeurs

- La classe est une description abstraite permettant d'instancier des objets.
- Un objet est une instance d'une classe.
- Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.

Opérateur `new`

- Opérateur de haute priorité
  - permet d'instancier des objets
  - en appelant une méthode particulière : le constructeur
- Création
  - fait appel à la machine virtuelle
    - pour obtenir l'espace mémoire nécessaire à la représentation de l'objet
  - appelle le constructeur
    - pour initialiser l'objet dans l'emplacement obtenu.
  - retourne une valeur
    - référence sur l'objet instancié (i.e. une adresse).
  - si new n'obtient pas l'allocation mémoire nécessaire
    - lève l'exception `OutOfMemoryError`.

Création d'un objet :

- Déclaration d'un objet ayant le type désiré :

```
MaClasse m;
```

- L'opérateur new créé une instance de la classe et l'associe à la variable :

```
m = new MaClasse();
```

- Il est possible de tout réunir en une seule déclaration :

```
MaClasse m = new MaClasse();
```

- Chaque instance d'une classe nécessite sa propre variable.
- Plusieurs variables peuvent désigner un même objet.
- les objets sont instanciés par allocation dynamique :
  - la variable m contient une référence sur l'objet instancié (i.e. contient l'adresse de l'objet qu'elle désigne)
    - pas d'opérations directement sur cette adresse comme en C.

Durée de vie :

- ne correspond pas forcément à la durée d'exécution du programme
- un objet passe par trois phases :
  - la *déclaration* de l'objet et l'*instanciation* grâce à l'opérateur new et le *constructeur*
  - l'*utilisation* de l'objet en appelant ses méthodes
  - la *suppression* de l'objet via l'utilisation d'un *destructeur*
- suppression d'un objet automatique
  - grâce à la machine virtuelle.
  - restitution de la mémoire libre par le 'récupérateur' (i.e. Garbage Collector)
  - Il n'existe pas de méthode `free` comme en C, ou `delete` comme en C++

Création d'objets identiques :

- Exemple :

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1;
```

- m1 et m2 contiennent la même référence
- ils pointent donc sur le même objet
  - une modification faite à partir de m1 ou m2 modifie l'objet.
- Pour créer une copie d'un objet, il faut utiliser la méthode clone()
  - créer un deuxième objet indépendant mais identique à l'original
  - méthode héritée de la classe Object
    - classe mère de toute les classes en Java.

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1.clone();
```

- m1 et m2 ne contiennent plus la même référence
  - ils pointent donc sur des objets différents.

Comparaison d'objets :

- Exemple :

```
Rectangle r1 = new Rectangle(100,50);
Rectangle r2 = new Rectangle(100,50);
if (r1 == r1) { ... } // vrai
if (r1 == r2) { ... } // faux
```

- l'opérateur == compare les références `r1` et `r2` ...
- égalité des variables de deux instances :
  - il faut munir la classe d'un méthode réservée à cette effet
  - la méthode `equals` héritée de `Object`.
    - et donc la méthode `hashcode()` de cet objet (Cf. TD)
- Comparaison des classes :
  - deux objets sont de la même classe ?
    - il faut utiliser la méthode `getClass()` de la classe `Object` dont toutes les classes héritent.

↪ Un objet particulier : null

- L'objet null est utilisable partout.
- n'appartient pas à une classe
- peut être utilisé à la place d'un objet de n'importe quelle classe
- peut être utilisé comme paramètre.
- null ne peut pas être utilisé comme un objet normal
  - il n'y a pas d'appel de méthodes
  - aucunes classes ne peuvent en hériter.
- si une variable référence un objet à null
  - le ramasse miette peut libérer la mémoire allouée à l'objet ...

↪ La variable this :

- permet de référencer dans une méthode l'instance de l'objet en cours d'utilisation.
- Utilisation :
  1. this est un objet égale à l'instance de l'objet dans lequel il est utilisé.

```
private int nombre;
public maclasse(int nombre) {
  nombre = nombre;
  // attribut  = variable en paramètre du constructeur
  // il est préférable d'écrire
  this.nombre = nombre
}
```

  2. this peut être utilisé par un objet pour appeler une méthode en se passant lui même en paramètre de l'appel
     - Exemple en IHM

↪ Instance d"un objet

- `instanceof()`
  - Détermine la classe de l'objet passé en paramètre

```
void testClasse(Object o) {
  if (o instanceof MaClasse )
  System.out.println("o est une instance de la classe MaClasse");
  else System.out.println("o n'est pas un objet de la classe MaClasse");
}
```

↪ Classe (i.e. type) d"un objet

- `getClass()`
  - Affichage du type

```
system.out.println(monObjet.getClass().toString());
```

↪ Garbage collector

- `system.gc()`
  - Force l'utilisation du ramasse-miettes

```
system.gc()
```

La déclaration d'un objet est suivie d'une initialisation :

- réalisée par une méthode particulière `text`,
- les variables ont ainsi une valeur de départ.
- Elle est systématiquement invoquée lors de la création d'un objet.

Le nom de constructeur doit obligatoirement correspondre à celui de la classe

Le constructeur n'est pas typé :

- il ne peut pas y avoir d'instruction return dans un constructeur
- le type est retourné implicitement : référence sur objet de la classe

Un constructeur peut être surchargé.

a définition d'un constructeur est facultative :

- Si non définie
  - la machine virtuelle appelle un constructeur par défaut
- Si explicitement défini
  - Java considère que le programmeur gère la création des constructeurs
    - plus de mécanisme par défaut (i.e. constructeur sans paramètres)
    - pour maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres.

Il existe plusieurs façon de définir un constructeur :

- le constructeur simple : ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.

```
public MaClasse() {}
```

- le constructeur avec initialisation fixe : créé un constructeur par défaut

```
public MaClasse() {
  nombre = 5;
}
```

- le constructeur avec initialisation des variables : pour spécifier les valeurs de données à initialiser il est possible de les passer en paramètres au constructeur

```
public MaClasse(int valeur) {
  nombre = valeur;
}
```

Exemple :

```
1   public class ClasseComplexe{
2
3     ...
4
5     /**
6     * Constructeur
7     * Construit un complexe nul
8     * (i.e. imComplexe = ReComplexe = 0.0f)
9     */
10    public ClasseComplexe(){
11      reComplexe = 0.0f;
12      imComplexe = 0.0f;
13    }
14
15    /**
16    * Constructeur
17    * Construit un complexe tq ReComplexe = a  et imComplexe = b
18    * @param a float : initialisation de la partie réelle
19    * @param b float : initialisation de la partie imaginaire
20    */
21    public ClasseComplexe(final float a, final float b){
22      reComplexe = a;
23      imComplexe = b;
24    }
25    ...
26  }
```

↪ Gestion de la mémoire

- Un destructeur permet d'exécuter des actions (i.e. du code) lors de la libération de l'espace mémoire occupé par l'objet.

- En java, les destructeurs appelés finaliseurs (`finalizers`) sont automatiquement appelés par le garbage collector.

- Pour créer un finaliseur, il faut redéfinir la méthode `finalize()` héritée de la classe `Object`.

IV - Concepts de la programmation objet

1 – Surcharge
2 – Redéfinition
3 – Encapsulation
4 – Héritage
5 – Polymorphisme
6 – Abstraction
7 – Interface

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents en nombre et/ou en type.

- attention, la sémantique de la méthode doit rester le même !
- Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments.
- Permet de simplifier l'interface des classes vis-à-vis des autres classes.
- Une méthode est donc surchargée lorsqu'elle exécute des actions similaires avec des types et/ou un nombre de paramètres transmis différent.

```
  ...
  public int calculValeur(int i) {
    return (i+10)*(i-3);
  }

  public float calculValeur(float f) {
    return (f+10)*(f-3);;
  }
  ...
}
```

La redéfinition d'une méthode permet de définir un nouveau comportement pour une méthode dans un contexte particulier

- Une redéfinition doit conserver la signature de la méthode existante :
  - nombre et types de paramètres
  - type de retour
  - les exceptions pouvant être propagées
- La méthode redéfinie est préfixée avec l'annotation `@Override`
- Le compilateur choisi la méthode en fonction du type de l'instance sur lequel elle est appliquée.
- exemple : redéfinition de la méthode `toString`

```
  private String nom;
  private String prenom;
  ...
  @Override
  public String toString() {
    return nom + " " + prenom;
  }
  ...
}
```

L'encapsulation permet de sécuriser l'accès aux données d'une classe

- les attributs déclarés `private` à l'intérieur d'une classe ne peuvent être accédés et modifiés que par des méthodes définies dans la même classe.
- Si une autre classe veut accéder aux données elle doit le faire par l'intermédiaire de méthodes de la classe prévue à cet effet : les accesseurs
- Accesseur :
  - méthode publique qui donne l'accès à une variable d'instance privée. Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un en écriture.
  - par convention, les accesseurs en lecture ("getters") commencent par get et les accesseurs en écriture ("setters") commencent par set.

```
private int valeur = 13;

public int getValeur(){
  return(valeur);
}
public void setValeur(int val){
  valeur = val;
}
...
```

↪ Définition

> Technique offerte par les langages de programmation pour construire une classe à partir d'une (ou plusieurs) autre(s) classe(s) en partageant ses attributs et opérations

- si une classe B hérite d'une classe A, B est vu comme un sous-type du type défini par A
  - Représente la relation ≪ Est Un ≫
  - Exemple : classe `VehiculeTerrestre` et classe `Moto`
    - une `Moto` *est un* `VehiculeTerrestre`
    - l'ensemble des `Moto` est inclus dans l'ensemble des `VehiculeTerrestre`
- la classe fille hérite des attributs et des méthodes de sa classe mère (et donc de tous ses ancêtres)
- la classe fille peut :
  - modifier ou redéfinir les méthodes héritées
  - ajouter de nouvelles méthodes

↪ Intérêt

- Spécialisation et enrichissement
- Redéfinition
- Factorisation (i.e. réutilisation de code)
- Notion de sous-type

↪ Syntaxe java

- utilisation du mot clé `extends`
- héritage simple
  - on n'hérite d'une et une seule classe
- si rien n'est précisé
  - le compilateur considère la classe `Object` comme classe parent ou classe mère.
- invoquer une méthode d'une classe parent
  - nom de la méthode préfixée par `super`.
- appel du constructeur de la classe parent
  - utilisation de `super` comme méthode :

```
super(paramètres)
```

- Java oblige dans un constructeur d'une classe fille à faire appel explicitement ou implicitement au constructeur de la classe mère.

↪ Visibilité

Accès aux propriétés héritées :

- Les variables et méthodes définies avec le modificateur d'accès `public` restent publiques lors de l'héritage
- Une variable d'instance définie avec le modificateur `private` est bien héritée, mais :
  - elle n'est pas accessible directement
  - accessible via les méthodes héritées
- Si l'on veut conserver une protection semblable à celle obtenue par le modificateur `private`
  - utilisation du modificateur `protected`
  - la variable sera héritée dans toutes les classes descendantes qui pourront y accéder librement
  - elle ne sera pas accessible directement hors de ces classes

↪ Upcasting & Downcasting 1/2

Upcasting (Surclassement) : Soit une classe B héritant d'une classe A

- toute instance de B peut-être vue comme une instance de A
- Exemple :

```
VehiculeTerrestre vt;
vt = new Moto();
```

- Règle :
  - pour une référence d'un type donné, une valeur de correspondant à un objet de toutes sous-classes directes ou indirecte peut lui être affectée

Downcasting (Sous-classement) :

- opération inverse
- à expliciter obligatoirement
- Exemple :

```
VehiculeTerrestre vt = new VehiculeTerrestre();
Moto m;
m = (Moto) vt;
```

↪ Upcasting & Downcasting 2/2

Surclassement et messages :

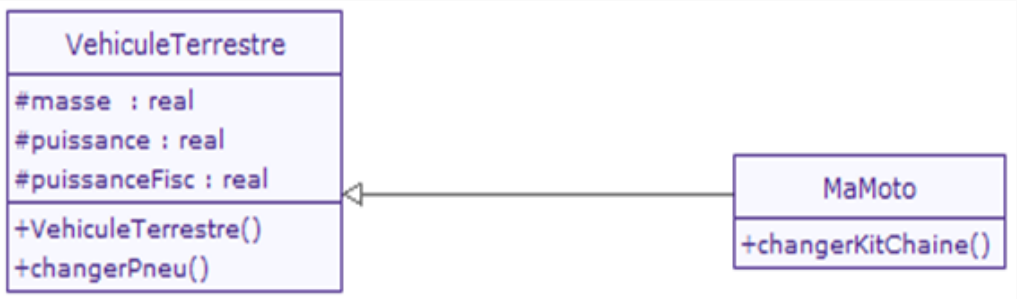
- un objet surclassé est considéré comme objet du type de la référence utilisée
  - → fonctionnalités restreintes à la classe du type de la référence

- Exemple :

```
Moto2 m=new Moto();
VehiculeTerrestre vt;
Vt=m;

Vt.changePneu();
m.changePneu();

m.changeKitChaine();
Vt.changeKitChaine(); // ERREUR !!!!
```



↪ Utilisation ...

De la bonne utilisation de l'héritage :

- Lors de la création d'une classe mère il faut tenir compte des points suivants :
  - la définition des accès aux variables d'instances, très souvent privés, doit être réfléchie entre `protected` et `private` ;
  - pour empêcher la redéfinition d'une méthode il faut la déclarer avec le modificateur `final`.

- Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas `final`, il faut envisager les cas suivant :
  - la méthode héritée convient à la classe fille : on ne doit pas la redéfinir
  - la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille :
    - il faut la redéfinir voir la surcharger.
    - en général, une redéfinition commencera par appeler la méthode héritée (via `super`) pour garantir l'évolution du code
  - la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.

↪ Définition

- un objet d'une classe peut-être *utilisé comme si il était instance d'une autre classe*
- la même méthode peut se *comporter différemment sur des objets issus de différentes classes de la hiérarchie d'héritage*

↪ Intérêt

- Développement plus rapide
- simplicité et organisation du code
- programme facilement extensible
- maintenance facilitée
- Exemple :
  - classe manipulant une liste de `VehiculeTerrestre`
  - peut contenir :
    - des objets `VehiculeTerrestre`
    - mais aussi des objets `Moto`
    - et tout objets issus d'une nouvelle classe héritant de `VehiculeTerrestre`

↪ Mise en œuvre

Réalisation :

- Héritage + Redéfinition + Upcasting

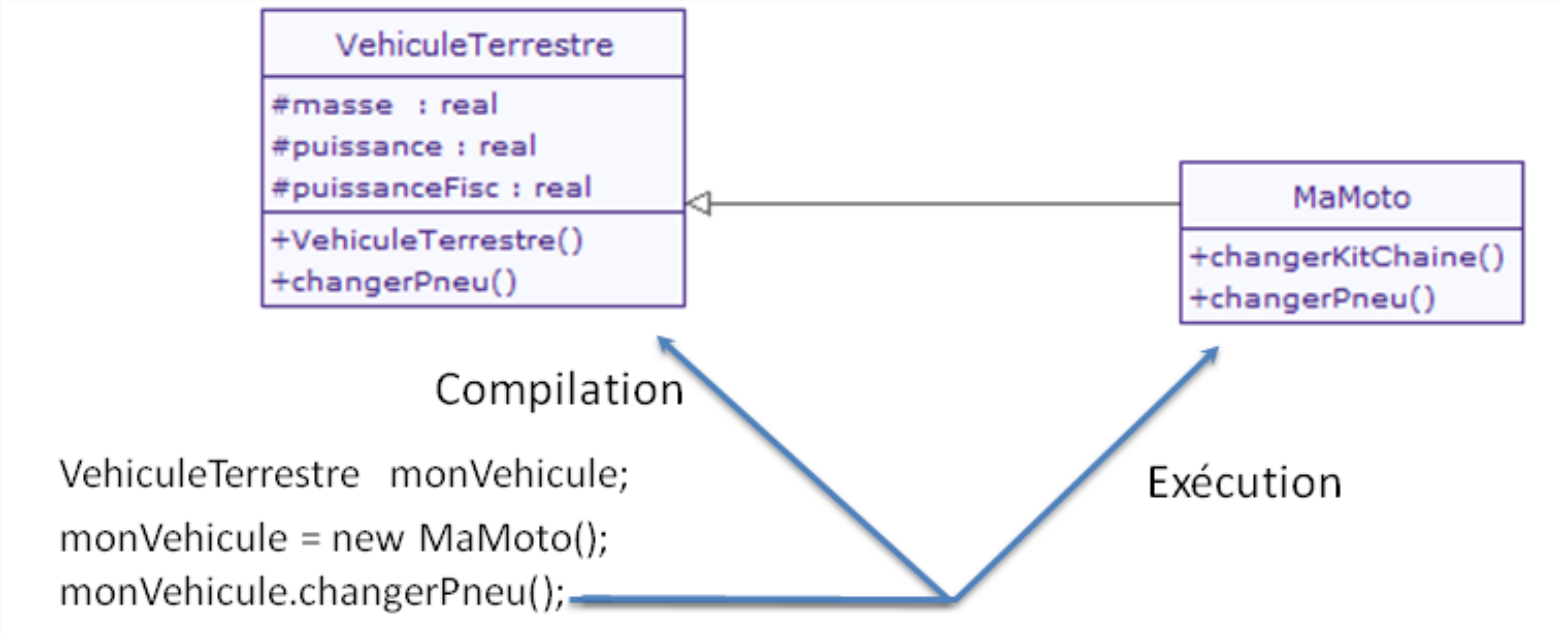
Fonctionnement :

- Vérification à la compilation :
  - existence de la méthode dans la classe de la référence utilisée

Lien dynamique :

- exécution du code présent dans la méthode de l'objet effectivement pointée par la référence
- On parle de :
  - liaison tardive
  - ou lien dynamique
  - ou dynamic binding
  - ou late-binding
  - ou run-time binding

↪ Exemple


VehiculeTerrestre
#masse : real
#puissance : real
#puissanceFisc : real
+VehiculeTerrestre()
+changerPneu()
MaMoto
+changerKitChaine()
+changerPneu()
Compilation
Exécution
VehiculeTerrestre monVehicule;
monVehicule = new MaMoto();
monVehicule.changerPneu();

↪ Définition

Objectif :

- Factorisation de code
- Généricité

Solution basique (et sale ...) :

- Utilisation de la classe `Object`
- Upcasting
- Avantage :
  - pas de limitation
- Oui mais ...
  - Pas de vérification
  - Pas assuré de n'avoir que des objets respectant un modèle, ou une sorte de contrat

Deux solutions évoluées (et propres ...) :

- Classe abstraite
  - abstraction de la structure de données
- Interface
  - abstraction des fonctionnalités

↪ Mise en œuvre

Classe abstraite :

- Définie par le mot clé `abstract`
- Une classe abstraite est non instanciable
- Le type défini par une classe abstraite est utilisable
  - grâce au polymorphisme
- Doit posséder une descendance concrète
  - sinon inutile

Méthode abstraite :

- Définie par le mot clé `abstract`
- Méthode ne possédant pas d'implémentation à ce niveau
- Une classe possédant au moins une méthode abstraite est forcément abstraite
  - l'inverse n'est pas vrai

↪ Utilisation

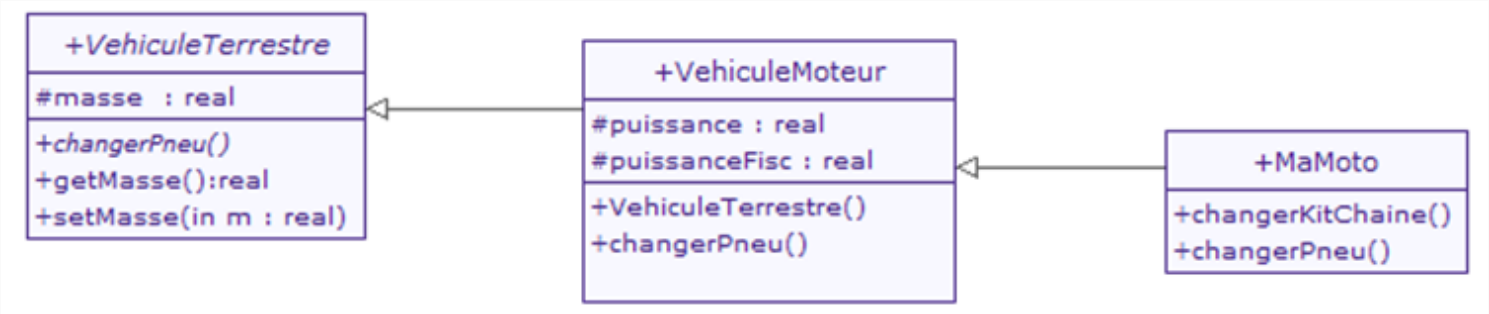
Classe fille d'une classe abstraite :

- Doit implémenter toutes les méthodes abstraites de la classe mère pour être *concrète*
- Sinon elle est elle-même abstraite

Utilisation des classes abstraites :

- regrouper certaines caractéristiques communes à ses sous-classes
- définir un comportement minimal commun

Exemple :



↪ Définition

- collection de méthode utilisée pour spécifier un service offert par une classe
- classe abstraite sans attributs et dont toutes les méthodes sont abstraites
  - doivent être absolument toutes réalisées (i.e. implémentées)
- une interface est un type !

↪ Réalisation et utilisation

- utilisation du mot clé `implements`
- Exemple :
  - Une interface est implicitement déclarée avec le modificateur `abstract`.

```
1       interface AfficheType {
2         void AfficheType();}
3
4       class Personne implements AfficheType {
5         public void afficheType() {
6           System.out.println("Je suis une personne");
7         }
8       }
9
10      class Voiture implements AfficheType {
11        public void afficheType() {
12          System.out.println("Je suis une voiture");
13        }
14      }
```

↪ Interfaces multiples

Une classe peut 'implémenter' plusieurs interfaces :

- nom des interfaces séparés par une virgule
  - permet de réaliser un héritage multiple (portant uniquement sur les méthodes) pour la classe

Réalisation : définition

- peut définir des constantes et méthodes abstraites
- une interface peut posséder des sous interfaces (i.e. héritage)
  - utilisation de `extends`
  - dans ce cas la classe implémentant cette interface doit définir
    - toutes les méthodes de l'interface
    - toutes les méthodes dont cette interface a hérité
- héritage multiple pour les interfaces
  - noms séparés par une virgule

Intérêt :

- grande souplesse de codage
- grande évolutivité du code
- augmente l'indépendance des différentes parties de code