
Programmation Java

Rafik Abbes - ISIMG © 2023



rafik.abbes@isimg.tn

Chapitre 1: Introduction

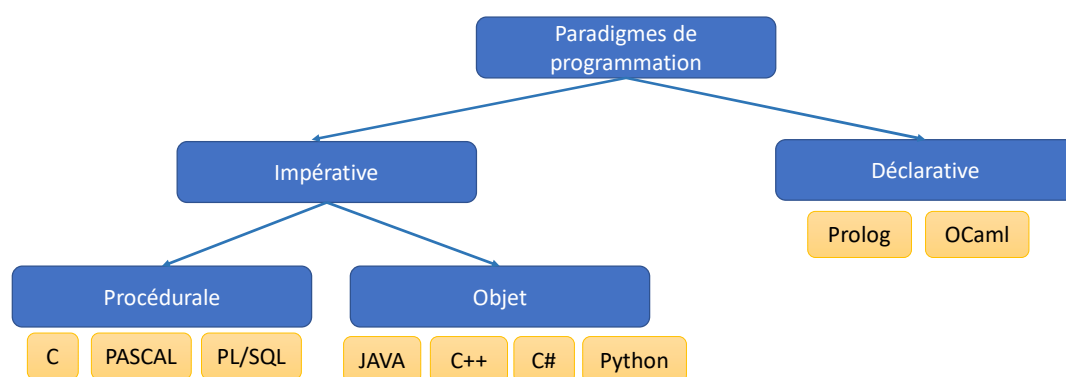
Qualité des produits logiciels
Paradigmes de programmation procédurale vs. Orientée Objet
Langage Java
Premier programme Java
JDK, IDE et livres Java

Qualité des produits logiciels

Facteurs	Capacité fonctionnelle
	Efficacité
	Compatibilité
	Facilité d'utilisation
	Fiabilité
	Sécurité
	Maintenabilité
	Portabilité

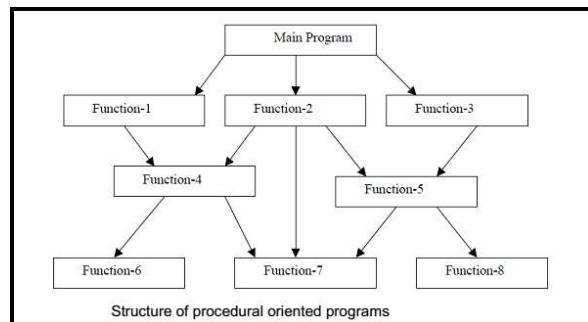


Paradigmes de programmation



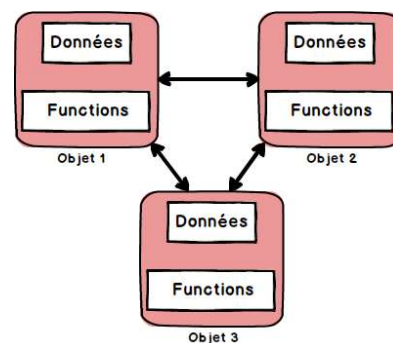
Programmation procédurale

- Décomposition du programme en **procédures** (ou fonctions).
- Données créées à l'intérieur des fonctions ou passées en paramètres.
- Priorité accordée aux fonctions et à la séquence des actions :
« Que doit faire mon programme ? »
- Pas de spécificateurs d'accès
- Réutilisation difficile



Programmation orientée objet

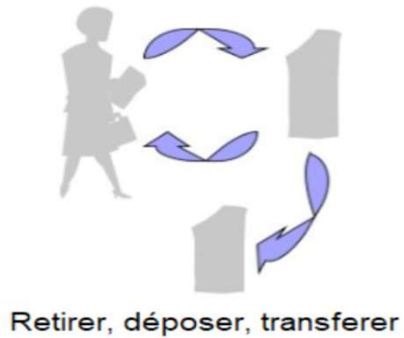
- Définition et interaction entre de briques logicielles appelées : **objets**
- Chaque objet regroupe des **données** et des **fonctions**
- Priorité accordée aux objets :
« De quoi doit être composé mon programme ? »
- Spécificateurs d'accès
- Réutilisation facile
- Code plus claire que la PP.



POO vs. PP

P. Procédurale

« Que doit faire mon programme ? »



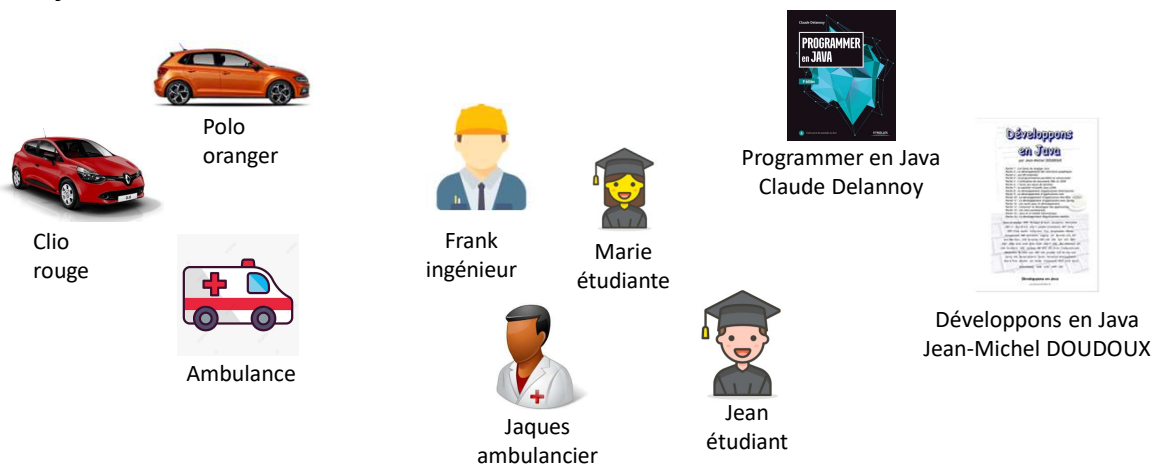
P. Orientée Objet

« De quoi doit être composé mon programme ? »



P.O.O : Qu'est qu'un objet ?

Objet : entité du monde réel



P.O.O : Qu'est qu'un objet ?

Un objet possède :

- Une **structure interne**
- Un **comportement**



Clio
rouge

Structure interne (attributs)

Modèle
Couleur
Puissance
Vitesse
Démarrée

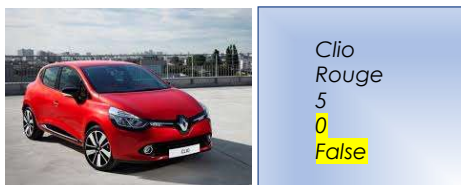
Comportement (méthodes)

Démarrer
Arrêter
Accélérer
Klaxonner
...

P.O.O : Qu'est qu'un objet ?

Un objet possède un **état** : valeurs des attributs à un instant donné
L'état d'un objet peut changer au cours du temps

Etat à l'instant t1



Etat à l'instant t2



P.O.O : Qu'est qu'une classe ?


Une **classe** : est une structure informatique permettant de décrire des objets similaires (ayant la **même structure interne** et le **même comportement**).



P.O.O : Qu'est qu'une classe ?

Exemple de classe en Java :

```
public class Voiture {  
    // définition des attributs  
    String modèle ;  
    String couleur ;  
    int puissance ;  
    int vitesse ;  
    boolean démarrée ;  
  
    // définition des méthodes  
    public void demarrer() {  
        démarrée = true ;  
    }  
    public void arreter() {  
        démarrée = false ;  
    }  
    public void accelerer(int v) {  
        vitesse = vitesse + v ;  
    }  
}
```

} Définition de la classe Voiture 

} Attributs

} Méthodes

4 Principes de la P.O.O

Abstraction



Eliminer les détails inutiles, ...

Encapsulation



Manipuler l'objet via ses méthodes accessibles ...

Héritage



Hériter et réutiliser le code d'une classe mère ...

Polymorphisme



Un même nom, plusieurs formes ...

Langage Java

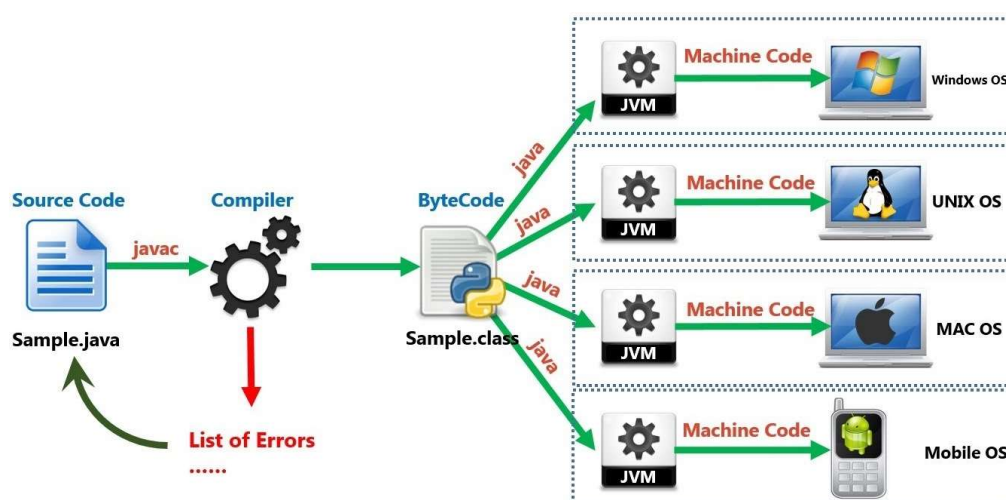
- Java est un langage orientée Objet
 - Développé chez SUN Microsystems, sous la direction de James Gosling.
 - Présenté en 1995
 - Racheté par Oracle en 2009.
 - Java est largement utilisé pour développer des applications variées (web, mobile, entreprise, ...)



Langage Java, caractéristiques

Interprété	Un programme Java n'est pas compilé code machine, mais en code intermédiaire appelé ByteCode . Lors de l'exécution, le ByteCode sera interprété à l'aide d'une machine virtuelle (JVM)
Portable	Le ByteCode est indépendant des plateformes. Il pourra être exécuté sur tous types de machines possédant un interpréteur de code Java.
Simple	<ul style="list-style-type: none">- Les auteurs de Java ont abandonné les notions mal comprises des autres langages de programmation telles que les pointeurs et l'héritage multiple.- Java se charge (presque) de restituer au système les zones mémoires inaccessibles et ce sans l'intervention du programmeur.
Robuste et sûr	<ul style="list-style-type: none">- Java détecte les erreurs d'invocation des méthodes.- Java est fortement typé : seules les conversions sûres sont automatiques
Orienté objet	Un programme Java est défini par des classes permettant la création des objets.
Multitâche	Java permet l'utilisation des threads qui sont des processus légers isolés.
Distribué	Il supporte les applications réseau

Compiler et exécuter un programme Java



Premier programme Java

Fichier source : `PremierProgramme.java`

```
// Commentaires
public class PremierProgramme {
    public static void main(String argst[]) {
        System.out.println("Hello World !");
    }
}
```

Mot-clé pour définir une classe

Nom de la classe
=
Nom du fichier source

Méthode principale
main

Instruction d'affichage
à l'écran

Compilation : `javac PremierProgramme.java`
Exécution : `java PremierProgramme`

Premier programme Java, remarques

- Une application minimale Java contient au moins une classe.
- Java est sensible à la casse : `Programme` \neq `PROGRAMME`
- Les caractères "{" et "}" marquent le début et la fin du bloc d'instructions à réaliser par la classe
- La méthode main est obligatoire, c'est le point d'entrée au programme.

Premier programme Java, remarques

- La signature de la méthode main est :

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

- Cette méthode est statique : **static**
- Ne renvoie rien : **void**
- Prend en paramètre un tableau args[] de type String.
 - args contiendra les arguments passés au programme lors de son exécution.

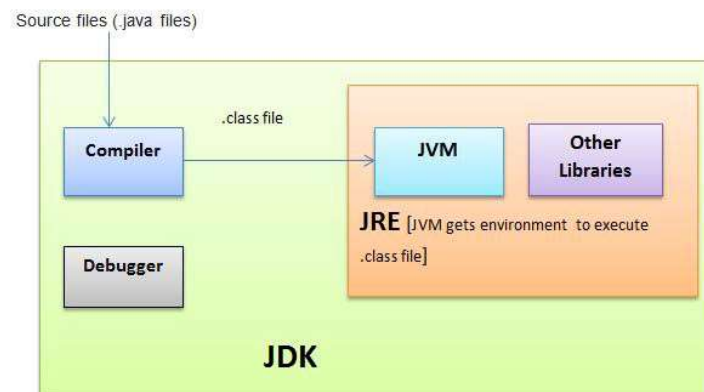
Exemple

Premier programme Java, remarques

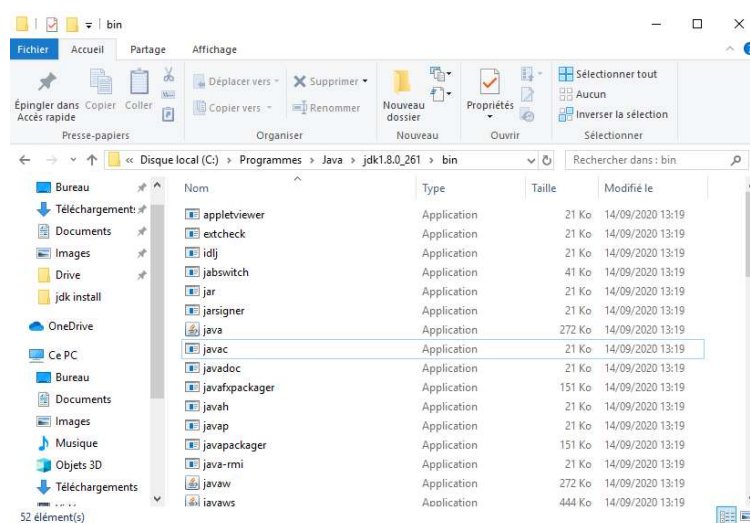
- Une méthode est une séquence d'instructions.
- Chaque instruction doit se terminer par un point-virgule ;
- `System.out.println()` est une instruction permettant d'afficher un message sur la sortie par défaut de votre machine (l'écran).
- L'indentation est ignorée du compilateur mais elle permet une meilleure compréhension du code par le programmeur.

Java Development Kit (JDK)

- JDK : ensemble des outils nécessaire pour développer et exécuter une application Java



Java Development Kit (JDK)

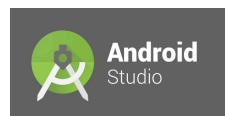


Environnement de développement intégrés (IDE)

- Les environnements de développements intégrés (IDE) regroupent dans un même outil la possibilité d'écrire du code source, d'exécuter et de déboguer le code.



JEdit



Livres Java

- Thinking in Java, Bruce Eckel, 4ème édition en 2006
- Algorithmique et programmation en Java, Vincent Granet, 2010
- Développons en Java (tutorial), Jean Michel DOUDOUX, version 2.1 en 2016
- Programmer en Java, Claude Delanno, ... 10ème édition en 2017

Développons
en Java



Chapitre 2:

Bases du langage Java

Commentaires, instructions, blocs, identificateurs, mots réservés

Types primitifs et types de référence

Opérateurs arithmétique, logique, de comparaison

Structures de contrôles conditionnelles et répétitives

Portée des variables

Commentaires

■ 3 types de commentaires

- Commentaire sur une seule ligne

```
// Commentaire sur une seule ligne
```

- Commentaire multi-lignes

```
/* Ce programme imprime la chaîne  
de caractère "Hello" à l'écran */
```

- Commentaire de documentation (javadoc)

```
/** Documentation de la classe */
```

Instructions, blocs

- Une instruction Java se termine par un point-virgule

```
int compteur = 0;
```

- Un bloc d'instruction est délimité par des accolades

```
{  
    String ch = args[0];  
    System.out.println("Hello " + ch);  
}
```

Identificateurs

Un **identificateur** est le nom utilisé pour désigner une classe, une méthode, une variable, ...

- ✓ Doit commencer par une **lettre**, **\$** ou **_**
- ✗ Interdiction d'utiliser les **mots réservés**

Mots réservés Java

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

Identificateurs : conventions d'écriture



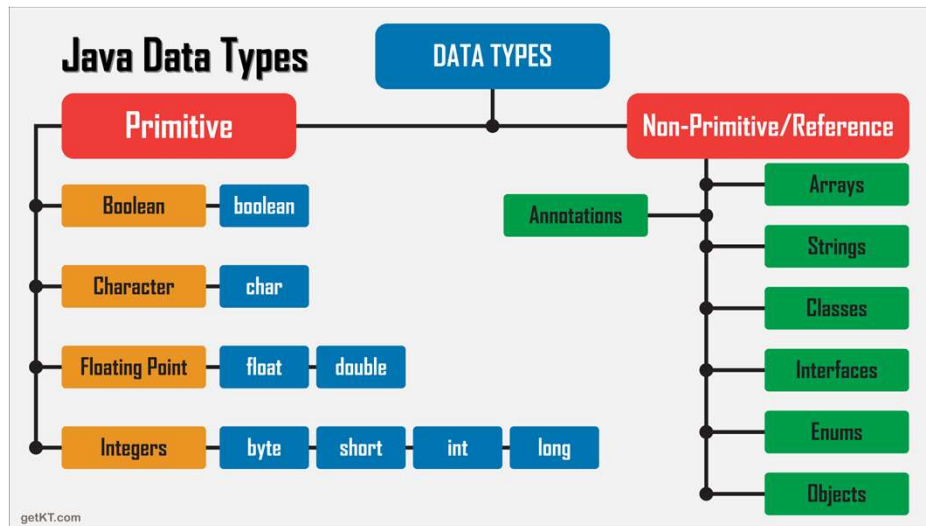
Upper Camel Case



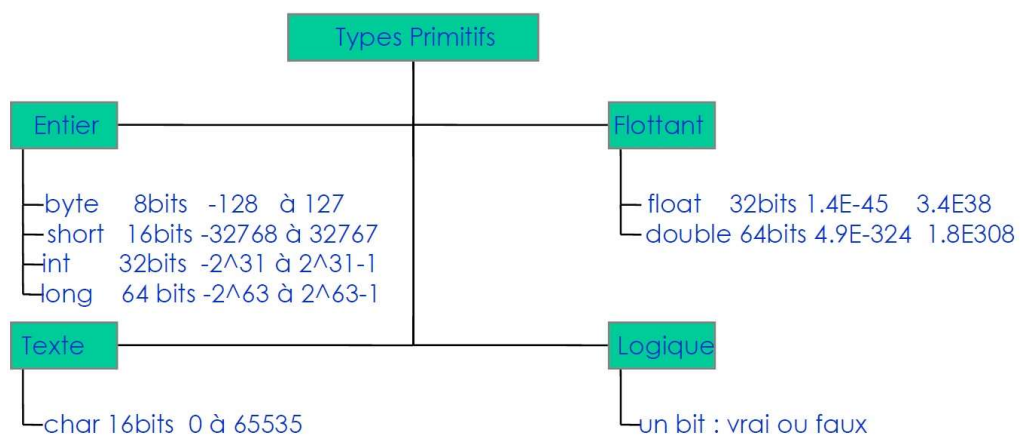
lower camel Case

	Ecriture	Exemples
Classe	Upper CamelCase	Etudiant CompteBancaire
Méthode	Verbes en lower camel Case	deposer getSolde setNom
Variable	Lower camel Case	compteur numEtudiant
Constante	En MAJUSCULE	PI MAX_TENTATIVES
Package	En minuscule	isimg.cours.poo

Types primitifs et types de référence





Types primitifs




Types primitifs

■ Déclaration, initialisation des variables

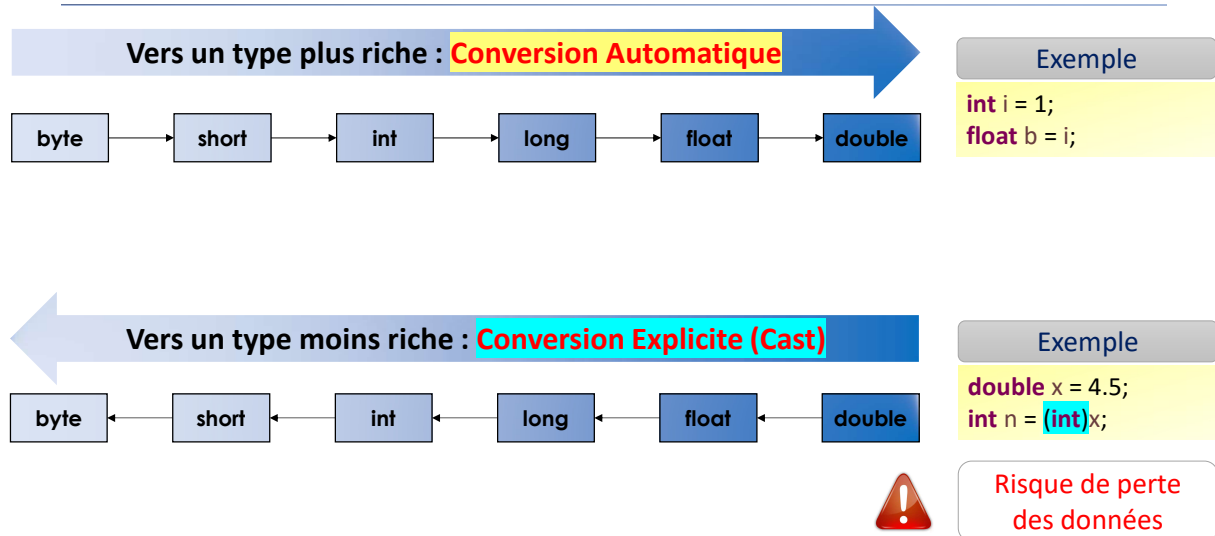
Déclaration	Déclaration et initialisation (1)	Déclaration et initialisation (2)
<pre>int i; int x,y,z; long bigNumber; char c; double d; float r; boolean found;</pre>	<pre>int i = 1; int x = 0, y = 0, z = 0; long bigNumber = 1000L; char c = 'A'; double d = 1234.59; float r = 7.5f; boolean found = true;</pre>	<pre>int i = 1; int a = i; float b = i; int j = b;</pre> <div> </div>

Types primitifs

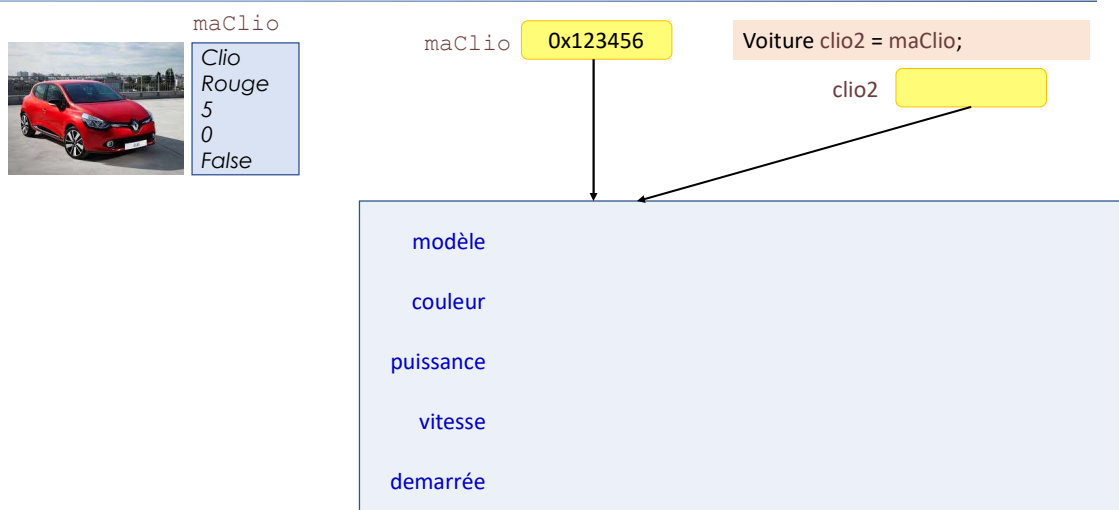
■ Déclaration, initialisation des variables

Déclaration	Déclaration et initialisation
<pre>final int x;</pre>	<pre>final double pi = 3.14; pi = 3.141 ;</pre> <div></div>

Types primitifs



Types de référence



Tableaux

- Un tableau est utilisé pour stocker une collection de variables de même type.
 - type primitif (int, double, boolean, ...)
 - type de référence

Déclaration

```
int numbers[];
```

Déclaration et allocation

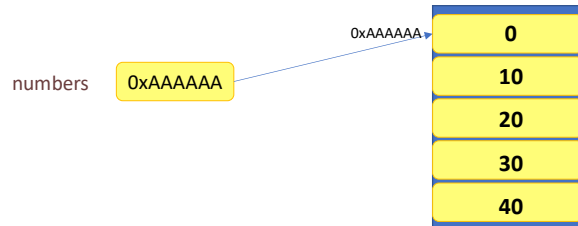
```
int numbers[] = new int[5];
```

Initialisation

```
numbers[0] = 0;  
numbers[1] = 10;  
numbers[2] = 20;  
numbers[3] = 30;  
numbers[4] = 40;
```

Déclaration, allocation et initialisation

```
int numbers[] = {0,10,20,30,40}
```

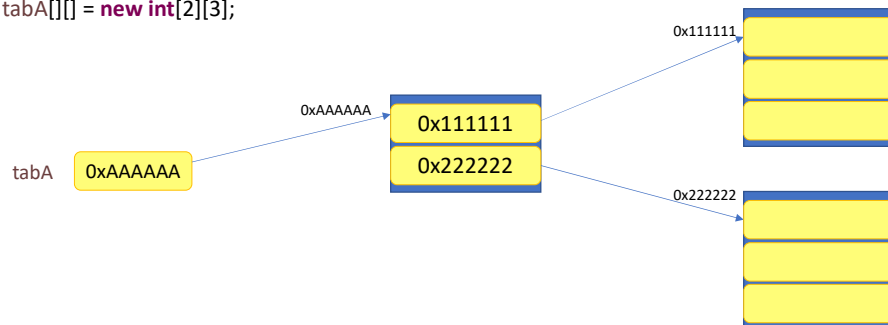


Tableaux à plusieurs dimensions

- Tableau à plusieurs dimensions, c'est un tableau de tableaux, ç.à.d, chaque case du premier tableau contient une référence vers un autre tableau.

```
// déclaration et allocation d'un tableau de 2 éléments (tableaux). Chaque élément est un tableau  
d'entier de taille maximale 3.
```

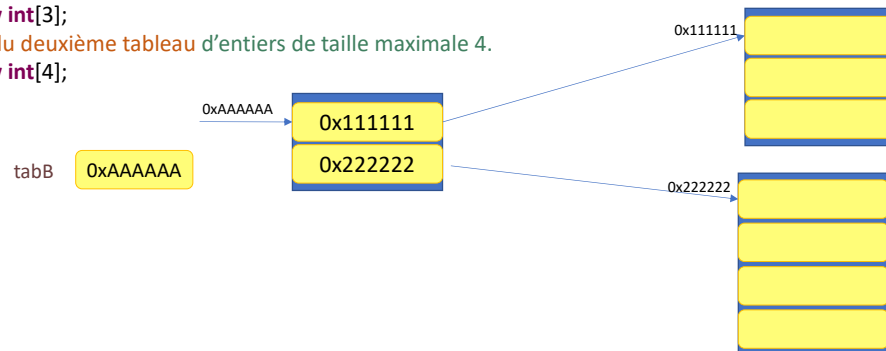
```
int tabA[][] = new int[2][3];
```



Tableaux à plusieurs dimensions

- Tableau à plusieurs dimensions, c'est un tableau de tableaux, ç.à.d, chaque case du premier tableau contient une référence vers un autre tableau.

```
// déclaration et allocation d'un tableau de 2 éléments (tableaux).  
int tabB[][] = new int[2][];  
// allocation du premier tableau d'entiers de taille maximale 3.  
tabB[0] = new int[3];  
// allocation du deuxième tableau d'entiers de taille maximale 4.  
tabB[1] = new int[4];
```



Arithmétique et opérateurs

- Règles de précédenes sur les opérateurs

Niveau	Symbole	Signification
1	()	Parenthèse
2	* / %	Produit Division Modulo
3	+ -	Addition ou concaténation Soustraction

Quel est le résultat de l'opération suivante: $4 + 3 * 2 + (5 / 2 + 3)$

Arithmétique et opérateurs

■ Opérateurs de comparaison

Opérateur	Exemple	Renvoie TRUE si
>	v1 > v2	v1 plus grand que v2
>=	v1 >= v2	Plus grand ou égal
<	v1 < v2	Plus petit que
<=	v1 <= v2	Plus petit ou égal à
==	v1 == v2	égal
!=	v1 != v2	différent

Arithmétique et opérateurs

■ Opérateurs logiques

Opérateur	Usage	Renvoie TRUE si
&&	expr1 && expr2	expr1 et expr2 sont vraies
&	expr1 & expr2	Idem mais évalue toujours les 2 expressions
	expr1 expr2	Expr1 ou expr2, ou les deux sont vraies
	expr1 expr2	idem mais évalue toujours les 2 expressions
!	! expr1	expr1 est fausse

Arithmétique et opérateurs

- L'opérateur d'affectation de base est =
- Il existe des opérateurs d'assignation qui réalisent à la fois une opération arithmétique et l'affectation

Opérateur	Exemple	Équivalent à
+=	expr1 += expr2	expr1 = expr1 + expr2
-=	expr1 -= expr2	expr1 = expr1 – expr2
*=	expr1 *= expr2	expr1 = expr1 * expr2
/=	expr1 /= expr2	expr1 = expr1 / expr2
%=	expr1 %= expr2	expr1 = expr1 % expr2

Arithmétique et opérateurs

- Opérateurs d'incrément et de décrémentation

Opérateur	Exemple	Explication
++ (postfixé)	x++	Évaluer x, puis incrémenter x
++ (préfixé)	++x	Incrémenter x, puis évaluer x
-- (postfixé)	x--	Évaluer x, puis décrémentation x
-- (préfixé)	--x	Décrémenter x, puis évaluer x

Structures de contrôle

Structure de contrôle	Syntaxe
Conditionnelle	if ... else
	switch ... case
Répétitive	for (; ;) { ... }
	while () { ... }
	do { ... } while ()

Structure conditionnelle if ... else

```
if(condition){  
    instructions;  
}else{  
    instructions;  
}
```

Exemple

```
int a=3;  
int b=4;  
int max;  
  
if(a>b){  
    max = a;  
}  
else {  
    max = b;  
}
```

L'opérateur raccourci ? :

```
max = (a>b?a:b);
```

Structure décisionnelle switch ... case

```
switch (variable){  
    case <val1> : instructions; break;  
    case <val2> : instructions; break;  
    ...  
    default: instructions; break;  
}
```

Exemple

```
int x = 11;  
switch (x) {  
    case 11: System.out.print( "Eleven"); break;  
    case 8 : System.out.print( "Eight" ); break;  
    default: System.out.print( "Other" ); break;  
}
```

Structure répétitive for

```
for(initialisation, condition, MAJ) {  
    instructions  
}
```

- **Initialisation** : S'exécute lorsque le programme rentre pour la 1ère fois dans la boucle
- **Condition** : elle doit être vérifiée pour exécuter une itération
- **MAJ** : Instruction exécutée chaque fois qu'une itération est terminée

Exemple

```
for (int i=0 ; i<5 ; i++) {  
    System.out.println("La valeur de i est : " + i);  
}
```


Structure répétitive while

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

Répéter les instructions tant que la **condition** spécifiée est Vrai

Exemple

```
int i = 0;  
while (i < 5) {  
    System.out.println("La valeur de i est : " + i);  
    i++;  
}
```

Structure répétitive do ... while

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

Répéter les instructions tant que la **condition** spécifiée est Vrai.
Les instructions seront exécutées au moins une fois.

Exemple

```
int i = 0;  
do {  
    System.out.println("La valeur de i est : " + i);  
    i++;  
}  
while (i < 5);
```

Instructions de branchement : break et continue

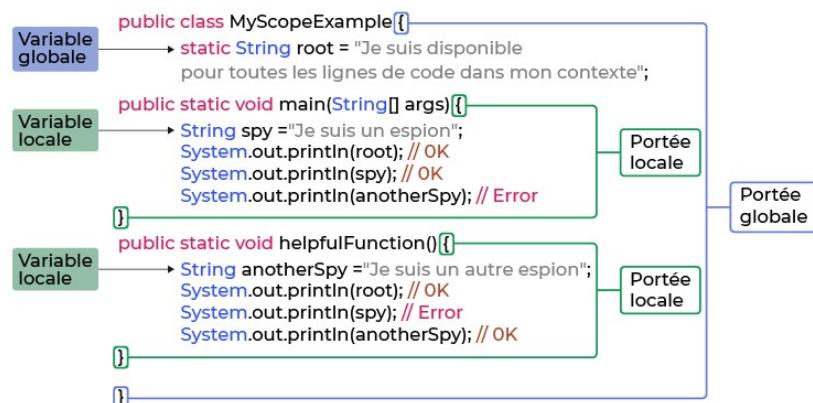
Dans une boucle

- **break** : achève immédiatement la boucle
- **continue** : ignore le reste des instructions et passe à l'itération suivante.

Exemple

```
for (i=0; i<10 ;i++){  
    if (i==5) continue; // Aller a l'itération suivante  
    if (i==7) break; // Sortir de la boucle  
    System.out.println("La valeur de i est: " + i);  
}
```

Portée d'une variable



Chapitre 3:

Programmation Objet avec

Java

Définition d'une classe

Constructeurs et instantiation d'objets

Notion d'encapsulation

Comparaison d'objets vs. Comparaison de références

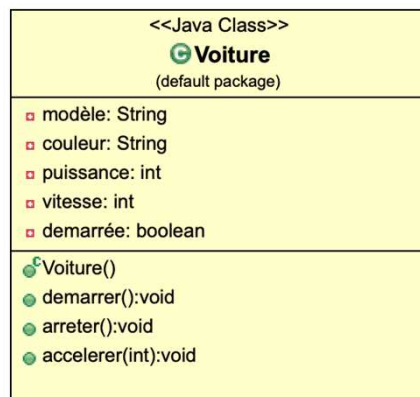
Surcharge de méthodes et de constructeurs

Attributs et méthodes statiques

Packages

Classe Java (représentation UML)

- Une classe Java est constituée d'un ensemble d'attributs et de méthodes



Représentation UML d'une classe

Classe Java (code)

```
public class Voiture {  
    // définition des attributs  
    private String modèle ;  
    private String couleur ;  
    private int puissance;  
    private int vitesse;  
    private boolean démarrée;  
  
    // définition des méthodes  
    public void demarrer() {  
        démarrée = true;  
    }  
    public void arreter() {  
        démarrée = false;  
        vitesse = 0;  
    }  
    public void accelerer(int v) {  
        if(démarrée)  
            vitesse = vitesse + v ;  
    }  
}
```

private : l'attribut n'est accessible que dans la classe courante.

public : la méthode est publique : elle est accessible depuis n'importe quelle classe.

Objet

- Un **objet** est une **instance** d'une classe
 - Admet une **valeur** pour chaque **attribut** déclaré dans la classe
 - Les valeurs des attributs définissent l'**état** d'un objet
 - Il est possible de lui appliquer une **méthode** définie dans la classe
 - Identifié et manipulé par sa **référence**

La classe Voiture va permettre d'**instancier** des **objets** de type Voiture et de leur appliquer à volonté les **méthodes publiques** : demarrer, arreter, accelerer

Instantiation d'un objet

```
/* Déclaration d'une variable de type Voiture. Elle réserve uniquement un emplacement pour une référence à un objet de type Voiture. */
```

```
Voiture maClio ;
```

```
/* new Voiture() permet la création d'un emplacement pour l'objet de type Voiture.
```

```
Cet objet est référencé par la variable maClio
```

```
*/
```

```
maClio = new Voiture();
```

new : est un opérateur Java qui permet l'instanciation d'un objet à partir d'une classe

maClio 0x123456

modèle	?
couleur	?
puissance	?
vitesse	?
demarrée	?

Initialisation d'un objet par une méthode

Ajoutons les méthodes `initialiser` et `afficher`

```
public class Voiture {  
    private String modèle ;  
    private String couleur ;  
    private int puissance;  
    private int vitesse;  
    private boolean demarrée;  
    ...  
    public void initialiser(String m, String c, int p) {  
        modèle = m;  
        couleur = c;  
        puissance = p;  
        vitesse = 0;  
        demarrée = false;  
    }  
    public void afficher() {  
        System.out.println("Je suis une voiture " + modèle + " de couleur " + couleur + " et de puissance " + puissance);  
    }  
}
```

Paramètres formels de la méthode

attributs de l'objet concerné par l'appel

Tester l'initialisation d'une voiture

```
public class TestVoitures {  
    public static void main(String[] args) {  
        // Déclaration et création d'une voiture  
        Voiture maClio = new Voiture();  
        // Appel de la méthode afficher appliquée sur la voiture maClio  
        maClio.afficher();  
        // Appel de la méthode initialiser appliquée sur la voiture maClio  
        maClio.initialiser("Clio", "Rouge", 5);  
        // Appel de la méthode afficher appliquée sur la voiture maClio  
        maClio.afficher();  
    }  
}
```



Il faut que l'utilisateur effectue l'initialisation de l'objet au moment opportun

Notion de constructeur

- La notion de **constructeur** permet d'automatiser le mécanisme d'initialisation d'un objet.
- Un constructeur est une "méthode spéciale", sans valeur de retour, portant le même nom que celui de la classe.
- Un constructeur peut disposer d'un nombre quelconque de paramètres (éventuellement aucun).
- Du moment où une classe dispose d'un constructeur défini, il n'est plus possible de créer un objet sans l'appeler.

Initialisation par un constructeur

Ajoutons un constructeur

```
public class Voiture {  
    private String modèle ;  
    private String couleur ;  
    private int puissance;  
    private int vitesse;  
    private boolean démarrée;  
  
    public Voiture(String m, String c, int p) {  
        modèle = m;  
        couleur = c;  
        puissance = p;  
        vitesse = 0;  
        démarrée = false;  
    }  
}
```

Paramètres du constructeur

Tester l'initialisation par un constructeur

```
public class TestVoitures {  
    public static void main(String[] args) {  
        Voiture maClio = new Voiture();  
        // La ligne précédente est ..... par le compilateur, car notre constructeur défini dans classe  
        Voiture a .....  
  
        // Création et initialisation via un constructeur  
        Voiture maClio = new Voiture(.....);  
        maClio.afficher();  
    }  
}
```

→ Je suis une voiture Clio de couleur Rouge et de puissance 5

Règles concernant les constructeurs

- Un constructeur ne renvoie jamais une valeur
- Une classe peut ne disposer d'aucun constructeur
- Un constructeur ne peut pas être appelé par un objet
- Un constructeur peut appeler un autre constructeur de la même classe, en utilisant le mot clé `this`

Initialisation ... Questions ?!

- En utilisant le constructeur par défaut,
 - l'attribut `puissance` a été initialisé à `0`
 - les attributs `couleur` et `modèle` ont été initialisés à `null`
 - Quelles sont les valeurs par défaut pour les autres types ?
- Que se passe-t-il si on initialise un attribut avec une valeur lors de sa déclaration dans la classe ?
- Et si on initialise ce même attribut dans un constructeur, quelle valeur sera prise en compte lors de l'instanciation ?

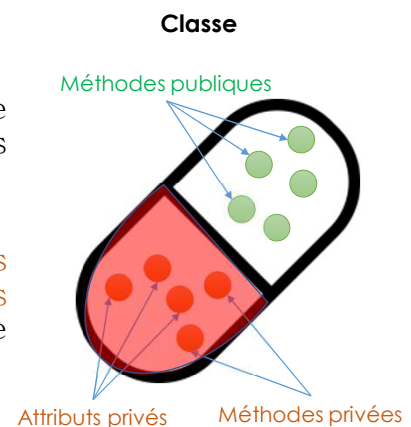
Ramasse-miettes (Garbage Collector)

- L'opérateur **new** permet, de créer un objet (allocation mémoire), cependant, on n'a pas un opérateur pour détruire un objet.
- Heureusement, Java peut gérer **automatiquement** la mémoire.
 - Java connaît à chaque instant le nombre de références à un objet donné.
 - Lorsqu'il n'existe plus aucune référence sur un objet, il est possible de libérer l'emplacement correspondant.



Éléments de conception des classes

- Une bonne conception orientée objet s'appuie sur la notion **d'encapsulation**
 - Considérer une classe comme un ensemble de services définis par les entêtes de ses **méthodes publiques**
 - Le reste, c.à.d. les **attributs et méthodes privés** ainsi que le **corps des méthodes publiques**, n'a pas à être connu de l'utilisateur de la classe.



Méthodes d'accès et de modification: *Getters & Setters*

- Une méthode d'accès (*getter*) renvoie la valeur d'un attribut
- Une méthode de modification (*setter*) *modifie* la valeur d'un attribut, *elle ne renvoie rien*
- Souvent, on nomme ces méthodes sous la forme *getToto* et *setToto* en respectant le lowerCamelCase

Toto : nom de l'attribut

Getters & Setters : exemple

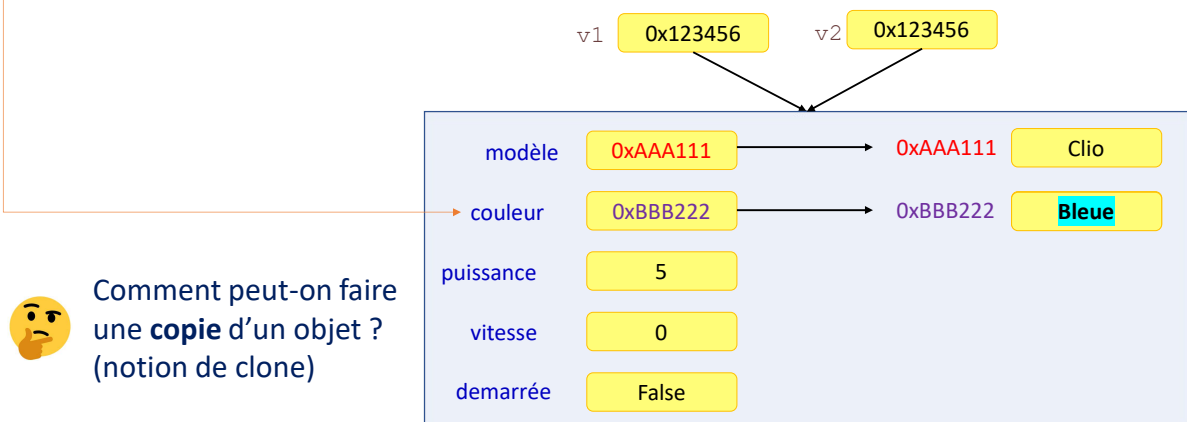
```
public class Voiture {  
    // Attributs  
    private String modèle ;  
    private String couleur;  
    private int puissance;  
    private int vitesse;  
    private boolean démarrée;  
  
    // Constructeurs  
    ...  
    // Méthodes :  
    // (1) Méthodes d'accès : getters  
    public String getCouleur() {  
        return couleur;  
    }  
    public int getPuissance() {  
        return puissance;  
    }  
  
    // (2) Méthodes setters  
    public void setCouleur(String c) {  
        couleur = c;  
    }  
    public void setPuissance(int p) {  
        puissance = p;  
    }  
  
    // (3) Autres méthodes  
    ...  
}  
// Fin de la classe Voiture
```

Affectation d'objets (1/2)

```
public class TestVoitures {  
    public static void main(String[] args) {  
        Voiture v1 = new Voiture("Clio", "Rouge", 5);  
        Voiture v2 = v1;  
        v2.setCouleur("Bleue");  
        v1.afficher();  
    }  
}
```

Affectation d'objets (2/2)

```
Voiture v1 = new Voiture("Clio", "Rouge", 5);  
Voiture v2 = v1;  
v2.setCouleur("Bleue");
```



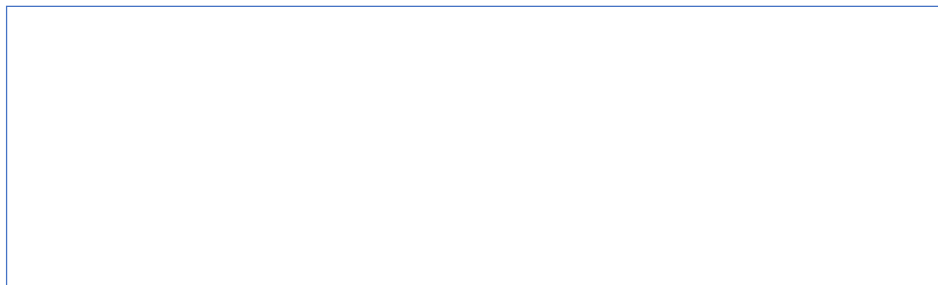
Comparaison de références

- Les opérateurs `==` et `!=` s'appliquent sur des références.
 - Intérêt limité, car la comparaison porte sur les références, et non pas sur les valeurs des attributs
 - `a==b` est vrai uniquement si `a` et `b` font référence à un seul objet.

```
public class TestVoitures {  
    public static void main(String[] args) {  
        Voiture v1 = new Voiture("Clio", "Rouge", 5);  
        Voiture v2 = new Voiture("Clio", "Rouge", 5);  
        if(v1==v2)  
            System.out.println("v1 et v2 référencent le même objet");  
        else  
            System.out.println("v1 et v2 réf. deux objets différents");  
    }  
}
```

Comparaison d'objets

- `v1` et `v2` de l'exemple précédant font références à deux objets égaux.
- Comment tester l'égalité de deux objets ?



Attributs statiques

Java permet de définir des attributs de classe (statiques)

```
public class B {  
    static int x ;  
    double y;  
}  
  
// dans la classe Test  
public static void main(String args[]) {  
    B b1 = new B();  
    B b2 = new B();  
}
```

Un attribut **statique** existe en **un seul exemplaire** quelque soit le nombre d'objets de la classe

Attributs statiques – Exemple

```
class Obj {  
    private static int nb = 0; // initialisation de nb  
    public Obj() {  
        System.out.println("obj créé");  
        nb++;  
        System.out.println("Il y en a maintenant " + nb);  
    }  
}  
  
public class TesObj{  
    public static void main(String[] args){  
        System.out.println("main 1");  
        Obj a = new Obj();  
        System.out.println("main 2");  
        Obj b;  
        System.out.println("main 3");  
        b = new Obj();  
        Obj c= new Obj();  
    }  
}
```

Méthodes statiques

- Une méthode statique est une méthode qui effectue un rôle indépendamment d'un objet quelconque.
 - Elle utilise (agit) uniquement (sur) des attributs statiques.

```
public class A {  
    static int x;  
    double y;  
  
    public static void f() {  
        System.out.println(x);  
        //System.out.println(y); Interdit !  
    }  
}
```

Méthodes statiques – Exemple

```
class Obj {  
    private static int nb = 0;  
    public Obj() {  
        System.out.println("objet créé");  
        nb++;  
    }  
    public static int nbObjets() {  
        return nb;  
    }  
}  
  
public class TesObj{  
    public static void main(String[] args){  
        System.out.println(Obj.nbObjets());  
        Obj a = new Obj();  
        Obj b = new Obj();  
        Obj c = new Obj();  
        System.out.println(Obj.nbObjets());  
    }  
}
```

Utilisations des méthodes statiques

- Permettre aux objets d'une classe de disposer d'informations collectives : exemple `static int nbObjets()`
- Fournir des services n'ayant signification que pour la classe même (`nom de classe`, `auteur`, `numéro de version`, ...)
- Regrouper au sein d'une classe des fonctionnalités ayant un point commun et qui ne sont pas liés à un objet donné
 - Exemple : la classe `Math` regroupe des fonctions de classe telles que `sqrt`, `sin`, `cos`, ...

Surcharge des méthodes

- La surcharge d'une méthode permet de `définir plusieurs fois une même méthode` avec `des paramètres différents`.
- A la rencontre d'un appel donné, le compilateur choisit la meilleure `méthode acceptable`.
 - Elle dispose du nombre de paramètres voulu
 - Le type de chaque paramètre effectif soit compatible par affectation avec le type du paramètre formel.

Surcharge des méthodes

```
public class Voiture {
    // définition des attributs ...
    ...
    // définition des méthodes
    public void accelerer(int v) {
        if(demarrée)
            vitesse = vitesse + v ;
    }
    public void accelerer() {
        if(demarrée)
            vitesse = vitesse + 10 ;
    }
    public void afficher() {
        System.out.println("Je suis une " + modèle + " " + couleur + " je roule à " + vitesse + "
        km/h");
    }
}
```

Surcharge des méthodes

```
public class TestVoitures {
    public static void main(String[] args) {
        Voiture v1 = new Voiture("Clio", "Rouge", 5);
        v1.demarrer();
        v1.afficher();
        v1.accelerer();
        v1.afficher();
        v1.accelerer(20);
        v1.afficher();
    }
}
```


Méthodes et échange des données

- Java utilise le mode de passage de paramètre par valeur
 - Une méthode reçoit une **copie** de la **valeur du paramètre effectif**.
 - Si ce **paramètre effectif** est de type **primitif**, sa copie peut être modifiée, sans que cela n'ait une incidence sur la valeur du paramètre effectif.
 - Si ce **paramètre effectif** est de type **référence**, la méthode travaille sur la **copie de cette référence**, **elle peut donc modifier l'objet concerné**.

Utilisation du mot clé this

- **this** peut être utilisé pour
 - appeler un constructeur à partir d'un autre constructeur
 - écrire les constructeurs de manière plus pratique sans avoir à donner des nouveaux noms aux paramètres.

```
public Voiture(String m, String c, int p){  
    modèle = m;  
    couleur = c;  
    puissance = p;  
}
```



```
public Voiture(String modèle, String couleur, int puissance) {  
    this.modèle = modèle;  
    this.couleur = couleur;  
    this.puissance = puissance;  
}
```

Paquetages « packages »

- Un package est un regroupement logique d'un ensemble de classes
 - Ex. `java.io` regroupe tout ce qui concerne les Entrées/Sorties
- Les classes d'un package sont sauvegardées dans un même répertoire
 - A chaque classe son fichier, à chaque package son répertoire
 - `java.awt.Point` -> `java/awt/Point.class`

```
package example;  
// La classe A fait partie du package example  
public class A {  
    ...  
}
```

Paquetages « packages »

- Si la déclaration du package est omise, la classe sera contenue dans le package par défaut.
- Les packages sont organisés en arborescence.
 - La désignation d'un package s'effectue en donnant le chemin sous la forme pointée

■ Ex. `isimg.lsim.progjava`;

Paquetages « packages »

- Il existe deux façons pour utiliser la classe `A` appartenant au package `example`

- Préfixer le nom de la classe par le nom du package

```
example.A a1 = new example.A();
```

- Importer la classe (ou les classes) du package au début du fichier source

```
import example.A;
// ou import example.* pour importer toutes les classes du package example

// la classe Test appartient au package par défaut
public class Test {
    public static void main(String args[]) {
        A a1 = new A();
    }
}
```

Paquetages « packages »

- Le package `java.lang` est automatiquement importé par le compilateur

- Ce qui permet d'utiliser les classes standards : `Math`, `System`, `Integer`, ... sans introduire l'instruction `import`

- Une classe déclarée `public` est visible depuis l'extérieur du package qui les contient

- Sans le mot clé `public`, la classe n'est accessible qu'aux classes du même package

- L'absence du mot clé `private` ou `public` pour une méthode, veut dire que cette méthode n'est accessible qu'à partir d'une classe du même package

Chapitre 4:

Héritage, polymorphisme, classes abstraites et interfaces

Notion de réutilisation en Java : par composition vs. par héritage

Héritage

Polymorphisme

Classes abstraites

Interfaces

Réutilisation des classes en Java

- Une des caractéristiques de Java est la **réutilisation du code**
- Deux manière pour réutiliser le code
 - Par **composition** : la nouvelle classe se compose d'objets de classes existantes.
 - Par **héritage** : la nouvelle classe est un type d'une classe existante. On prend la forme d'une classe existante et on lui ajoute du code (nouvelles propriétés et méthodes ou redéfinition)

En créant de nouvelles classes, au lieu de les créer du zéro, on utilise les classes construites et testées par d'autres développeurs (sans modifier le code existant).



Réutilisation par composition

- On se dispose de la classe Point (déjà codée) pour créer des objets points dans un espace à 2 dimensions.
- Nous souhaitons définir une nouvelle classe pour créer des cercles.
 - Un cercle est caractérisé par
 - Son centre (objet de type Point)
 - Son rayon de type float

Réutilisation par composition

- La classe Point

```
public class Point {  
    private int x;  
    private int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public void setY(int y) {  
        this.y = y;  
    }  
    public void setX(int x) {  
        this.x = x;  
    }  
    public String toString() {  
        return "le point (" + x + ", " + y + ")";  
    }  
}
```

Réutilisation par composition

```
public class Cercle {
    private Point centre;
    private float rayon;

    public Cercle() {
        centre = new Point(0, 0);
        rayon = 1;
    }

    public Cercle(int x, int y, float rayon) {
        centre = new Point(x, y);
        this.rayon = rayon;
    }

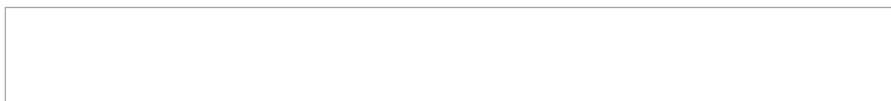
    public float getRayon() {
        return rayon;
    }

    public void setRayon(float rayon){
        this.rayon = rayon;
    }

    public String toString() {
        return "Cercle de centre " + centre.toString()
            + " et de rayon=" + rayon;
    }
}
```

Réutilisation par composition

```
public class TestCercles {
    public static void main(String[] args) {
        Cercle c1 = new Cercle();
        Cercle c2 = new Cercle(1,2, 4);
        Cercle c3 = c1;
        c3.setRayon(2);
        Cercle tab_cercles[] = {c1, c2, c3}; // un tableau de cercles
        for(int i=0; i<tab_cercles.length; i++) {
            System.out.println(tab_cercles[i].toString());
        }
    }
}
```



Réutilisation par héritage

■ L'héritage est un mécanisme qui facilite la **réutilisation** du code et la gestion de son évolution.

■ L'héritage définit une relation entre deux classes :

- une **classe mère** (ou **super-classe**)
- une **classe fille** (ou **sous-classe**) qui **hérite** de sa **classe mère**

■ Mise en œuvre de l'héritage :

```
public class Fille extends Mere {  
    ...  
}
```

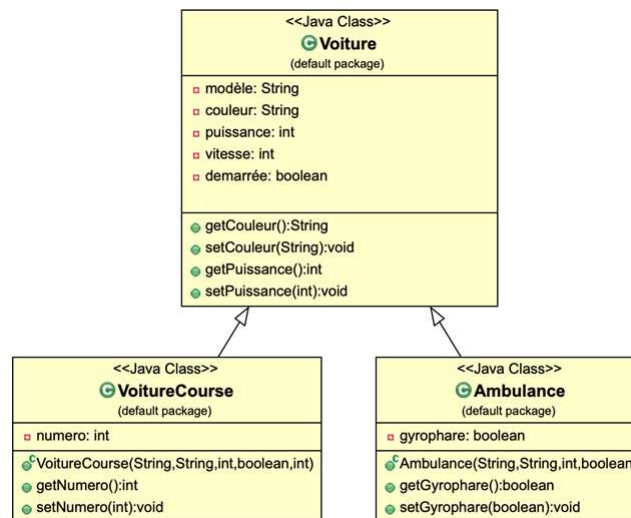
Réutilisation par héritage

■ L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de **super-classes** et de **sous-classes**

■ Une classe peut avoir plusieurs **sous-classes**.

■ Une classe ne peut avoir **qu'une seule classe mère** : il n'y a pas d'héritage multiple en Java.

Réutilisation par héritage



Réutilisation par héritage

- Les objets de **classe fille** héritent les attributs et méthodes de la **classe mère**.
 - Le constructeur de la **classe fille** doit prendre en charge l'intégralité de la construction de l'objet (**attributs hérités** ou **nouveaux**)
- Les **classes filles** peuvent redéfinir les attributs et les méthodes hérités.
 - Pour les attributs : déclarer un attribut sous le même nom avec un type différent (**usage peu courant**)
 - Pour les méthodes : redéfinir une méthode avec **la même signature** (même nom, même type de retour et mêmes paramètres)

Héritage et visibilité des membres

- Les membres (attributs et méthodes) définies avec le modificateur **public** sont toujours accessibles à travers la classe fille ou toute autre classe.
- Un attribut défini avec le modificateur **private** est hérité, **mais n'est pas accessible directement**.
 - Il est accessible par les méthodes héritées.
- Un attribut défini avec le modificateur **protected** sera hérité dans les classes filles qui **pourront y accéder librement***.

* Les classes du même package peuvent également accéder librement aux attributs **protected**

Modificateurs de visibilité

Modifieur	Class	Package	Subclasses	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

Héritage et accès aux membres

- Le mot clé **super** permet d'accéder aux membres de la super classe à partir de la classe fille.

super() appelle* le constructeur sans paramètre de la super-classe à partir du constructeur de la classe fille.

* Si un tel appel est fait, il doit obligatoirement s'agir de la 1ere instruction du constructeur

super.attribut permet l'accès à un attribut de la super classe.

super.methode() permet l'accès à la méthode de la super classe.

Héritage – exemple

- On se dispose maintenant des classes Point et Cercle.
- On souhaite créer des objets cercles colorés.
- Un cercle coloré ce n'est qu'un cercle auquel on ajoute une **couleur**.



Grace au mécanisme d'héritage, on peut réutiliser le code existant (classe Cercle), et n'ajouter que les nouvelles spécifications (ici, la **couleur**)

Héritage – exemple

```
public class CercleCouleur extends Cercle {
    private String couleur;

    public CercleCouleur(String couleur) {
        super();
        this.couleur = couleur;
    }
    public CercleCouleur(int x, int y, float rayon, String couleur) {
        super(x, y, rayon);
        this.couleur = couleur;
    }
    public String getCouleur() {
        return couleur;
    }
    public void setCouleur(String couleur) {
        this.couleur = couleur;
    }
}
```

Héritage – exemple

```
public class TestCercles {
    public static void main(String[] args) {
        Cercle c1 = new Cercle();
        CercleCouleur c2 = new CercleCouleur(1,2, 4, "Bleu");
        c2.setRayon(2);
        System.out.println(c2.toString());
        System.out.println(c2.getCouleur());
    }
}
```



Héritage – exemple

```
public class TestCercles {  
    public static void main(String[] args) {  
        Cercle c1 = new Cercle();  
        CercleCouleur c2 = new CercleCouleur(1,2, 4, "Bleu");  
        Cercle c3 = new CercleCouleur(0,0,3, "Blanc");  
        c2.setRayon(2);  
        System.out.println(c2.toString());  
        System.out.println(c2.getCouleur());  
        // System.out.println(c3.getCouleur());  
        System.out.println( ((CercleCouleur)c3).getCouleur() );  
        System.out.println(((CercleCouleur)c1).getCouleur());  
    }  
}
```

→ Erreur
→ Erreur

Redéfinition d'une méthode

```
CercleCouleur c2 = new CercleCouleur(1,2, 4, "Bleu");  
System.out.println(c2.toString());
```

→ Cercle de centre le point (1,2) et de rayon=2.0

Java permet de **redéfinir** une méthode héritée pour mieux s'adapter à l'objet de la **classe fille**.

La méthode redéfinie doit avoir la même signature.

```
// Redéfinition de la méthode toString dans la classe CercleCouleur  
public String toString() {  
    return super.toString() + " et de couleur " + couleur;  
}
```

→ Cercle de centre le point (1,2) et de rayon=2.0 et de couleur Bleu

Redéfinition d'une méthode



1. Peut-on modifier la méthode `toString()` pour afficher le message « *Cercle **Bleu** de centre le point (1,2) et de rayon=2.0* » lorsqu'elle est appliquée sur l'objet `c2`.
2. Peut-on redéfinir une méthode `private` ?
3. Si oui, peut-on changer augmenter la visibilité de la méthode redéfinie, (la rendre `public`)
4. Peut-on redéfinir une méthode `public` et diminuer sa visibilité (la rendre `private`)?

Polymorphisme

- Le polymorphisme est un concept extrêmement puissant en P.O.O qui complète l'héritage.

Héritage



Hériter et réutiliser le code d'une classe mère ...

Polymorphisme



Un même nom, plusieurs formes ...

- Il permet de manipuler des objets sans en connaître (tout à fait)
- Il exploite la relation "est un" ("is a" en anglais) induite par l'héritage
 - Exemple 1: Un **CercleCouleur** est un **Cercle**
 - Exemple 2: Un **Chat** est un **Animal**

Polymorphisme, exemple 1

■ Considérons les deux classes Cercle et CercleCouleur

■ Evidemment, on peut écrire :

```
Cercle c1 = new Cercle();  
CercleCouleur c2 = new CercleCouleur(1,2, 4, "Bleu");
```

o Un cercle Couleur est un Cercle , on peut écrire donc

```
c1 = ..... ✓  
// Cercle : type .....  
// CercleCouleur : type .....
```

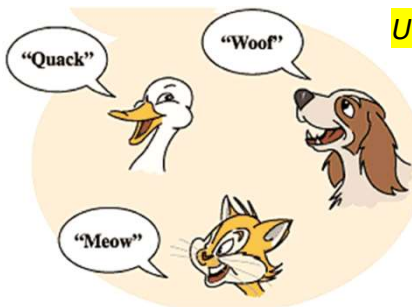
o On ne peut pas écrire l'inverse

```
c2 = ..... ✗
```

Polymorphisme, exemple 2

■ Considérons un tableau d'objets contenant, un canard, un chat et un chien.

```
Animal myObjs[] = {new Chien(), new Canard(), new Chat()};  
for(int i=0; i<3; i++) {  
    System.out.println(myObjs[i]. speak());  
}
```

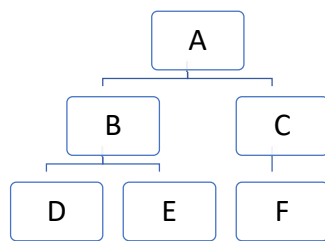


Un même appel de méthode ...

plusieurs comportements ...

- Les classes Chien, Canard et Chat héritent (extends) de Animal
- La méthode speak est définie dans la classe mère Animal
- La méthode speak est redéfinie dans les classes filles : Chien, Canard et Chat

Polymorphisme à plusieurs classes



Avec ces déclarations

A a; B b; C c; D d; E e; F f;

Les affectations suivantes sont légales

.....
.....
.....

Les affectations suivantes ne le sont pas

b = a; //
d = c; //
c = d; //

Règles de polymorphisme en Java

- **Compatibilité** : Il existe une **conversion implicite** des types lors de l'affectation. (ex. un CercleCouleur est un Cercle)
- **Liaison dynamique** : dans un appel de la forme x.f(...) où x est un objet de type T, le choix de f est déterminé ainsi:
 - **À la compilation** : la classe T ou l'une de ses ascendantes contient une méthode f **acceptable** à l'appel.
 - **À l'exécution** : on recherche la méthode f avec la signature voulue, à **partir** de la classe correspondante au **type effectif** de l'objet référencé. **Si cette classe ne comporte pas la méthode appropriée**, on remonte dans la hiérarchie jusqu'à ce qu'on retrouve une.

Conversions explicites de références

```
Cercle cer1 = new CercleCouleur(1,2, 4, "Bleu");  
CercleCouleur cer2 = cer1; /* erreur de compilation car on essaye d'affecter une  
référence de type Cercle dans une référence de type plus spécifique (CercleCouleur) */
```

```
Cercle cer1 = new CercleCouleur(1,2, 4, "Bleu");  
CercleCouleur cer2 = (CercleCouleur) cer1;
```

Conversion explicite (cast)

Conversions explicites de références

- La conversion explicite est une **conversion à risque**.
 - Dans l'exemple précédant, cer1 est de **type effectif CercleCouleur**, sa conversion vers le **type de référence CercleCouleur** se passe sans aucun problème lors de l'exécution.
 - **Mais attention**, si on essaie de convertir un **type effectif Cercle** vers un type de référence **CercleCouleur**, cela engendra une **erreur d'exécution**.
- Il est possible de vérifier le type effectif d'un objet donné en utilisant l'opérateur **instanceOf**

```
if (cer1 instanceof CercleCouleur)
```


Classes et méthodes finales

- Une méthode déclarée **final** ne peut pas être redéfinie dans une classe dérivée.
- Une classe déclarée **final** ne peut pas être dérivée. (on ne peut pas l'étendre avec **extends**)

Classes abstraites

- Une classe abstraite est une classe qui **ne permet pas d'instancier des objets**.
 - Elle peut contenir classiquement des méthodes et des attributs
 - On peut trouver des méthodes dites **abstraites** : c.à.d des méthodes contenant uniquement l'entête.

```
public abstract class A {  
    public void f() {  
        System.out.println("Here is f method");  
    };  
    public abstract void g(int n);  
}
```

```
A a ; // Déclaration d'un objet de type A, OK !  
a = new A(); // Instanciation interdite ! -> Erreur de compilation
```

Classes abstraites

- Si on dérive de A une classe B qui implémente la méthode abstraite g,

```
public class B extends A{  
    public void g(int n) {  
        // code d'implémentation de la méthode g ...  
    }  
}
```

on pourra alors instancier un objet de type B par `new B(...)` et même affecter sa référence à une variable de type A.

```
A a ; // Déclaration d'un objet de type A, OK !  
a = new B(); // Instanciation OK, car B n'est pas abstraite
```

Classes abstraites : règles

1. Dès qu'une classe comporte au moins une méthode abstraite, elle est abstraite
2. Une méthode abstraite ne peut pas être déclarée private
3. Une classe héritant d'une classe abstraite n'est pas obligée à implémenter toutes les méthodes abstraites de sa classe mère. Dans ce cas, elle reste simplement abstraite
4. Une classe héritant d'une classe non abstraite, peut être déclarée abstraite (déjà, implicitement, toutes les classes dérivent de Object)

Classes abstraites : Exemple

- Définition de classe abstraite Shape avec la méthode abstraite `perimeter()`;

```
class abstract Shape
{
    ...
    public Point getPosition() {
        return posn;
    }
    public abstract double perimeter();
    ...
}
```

```
class Circle extends Shape
{
    ...
    public double perimeter() { return 2 * Math.PI * r; }
}
```

```
class Rectangle extends Shape
{
    public double perimeter() { return 2 * (height + width); }
}
```

Classes abstraites : Exemple

- L'intérêt est d'exploiter le principe du polymorphisme
Dans la méthode main :

```
Shape[] shapes = { new Circle(2),
                  new Rectangle(2,3),
                  .....
                };
```

```
double sum_of_perimeters = 0;
for (int i = 0; i < shapes.length; i++)
    sum_of_perimeters += shapes[i].perimeter();
```

Interfaces

- Si considère qu'une classe abstraite n'implémentant aucune méthode et aucun attribut variable, on aboutit à la notion d'interface.
 - Une interface est en quelque sorte "une classe totalement abstraite"
 - On ne peut pas instancier une interface
 - On peut typer des données par le nom d'une interface
 - Les interfaces peuvent se dériver.
 - Une classe pourra implémenter plusieurs interface
 - La notion d'interface va se superposer avec celle de dérivation

Interfaces

- Dans la définition d'une interface, on peut trouver que des méthodes abstraites et des constantes.
- Une classe peut implémenter une interface donnée en utilisant le mot-clé **implements**

```
public interface I1 {  
    static final int MAX = 100;  
    void f(int n);  
    void g();  
}
```

```
public class A implements I1 {  
    public void f(int n) {  
        // Implémentation de la méthode f  
    }  
    public void g() {  
        // Implémentation de la méthode g  
    }  
}
```

Chapitre 5:

Classes de base

Classe Object

Classes enveloppes (Wrappers)

Les chaînes de caractères

La classe Vector

Classe Object

■ La classe **Object** est la classe racine de toutes les classes en Java.

■ Elle fournit des méthodes communes à toutes les classes

■ `public boolean equals(Object obj)` teste l'égalité de l'objet courant par rapport à l'objet passé en paramètre.

L'implémentation de base de cette méthode teste l'égalité des références.

■ `public int hashCode()` retourne le hash code de l'objet.

■ `protected Object clone()` pour dupliquer l'objet référencé.

■ `public final Class getClass()` chaque classe a une représentation à l'exécution, cette représentation est un objet de type `Class`.

■ `protected void finalize()` throws `Throwable` pour le ramasse-miettes.

■ Certaines méthodes doivent être redéfinies dans les classes filles pour fonctionner correctement

Classes enveloppes

- Les classes enveloppes (wrappers en anglais) permettent de manipuler des types primitifs comme des objets.
- Il existe des classes nommées Boolean, Character, Byte, Short, Integer, Long, Float et Double qui encapsulent des valeurs du type primitif correspondant.

Chaines de caractères : classe String

- Le type chaîne de caractère est un type de référence

- Déclaration

```
String maChaine ;
```

- Création

```
maChaine = "Bonjour";  
maChaine = new String("Bonjour");
```

- Déclaration et création:

```
String maChaine = "Bonjour";
```

<https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

Classe String : lenght, charAt, +

■ Longueur d'une chaîne de caractère

```
String Chaîne = "Bonjour";  
int n = maChaîne.length(); // n contient 6
```

■ Accès aux caractères d'une chaîne

```
char c = maChaîne.charAt(0); // c contient le caractère B
```

■ Concaténation de chaînes

```
String ch1 = "Hello";  
String ch2 = "Developers";  
String ch = ch1+ch2; // ch référence la chaîne "HelloDevelopers"  
String prix = 200 + "TND" // conversion et concaténation  
String txt = "bonjour\n" + "ligne2";
```

Classe String: rechercher dans une chaîne

Méthodes : indexOf et lastIndexOf

```
String ch = "bonjour"; int n;  
n = ch.indexOf('n'); // n vaut 2  
n = ch.lastIndexOf('o'); // n vaut 4  
n = ch.indexOf("jo"); // n vaut 3
```

Classe String: comparaison de chaines

Méthode equals()

```
String ch1 = "bonjour";
String ch2 = "bonjour";
String ch3 = "bon";
ch3 += "jour";

boolean b1 = ch1.equals(ch2); // b1 vaut true
boolean b2 = ch1.equals(ch3); // b2 vaut true
boolean b3 = ch1 == ch2; // b3 vaut false
boolean b4 = ch1.equalsIgnoreCase("BONJOUR"); // b4 vaut true
```

Classe String: modification de chaines

replace() : Remplacement de caractère(s)

```
String ch1 = "Bonjour";
String ch2 = ch1.replace("jour", " weekend"); // ch2 référence la chaine "Bon weekend"
```

substring() : extraction de sous-chaine

```
String ch3 = ch1.substring(3); // ch3 contient jour
String ch4 = ch2.substring(4, 8); // ch4 contient week
```

toUpperCase(), toLowerCase

```
String ch5 = ch1.toUpperCase(); // ch5 contient BONJOUR
String ch6 = ch1.toLowerCase(); // ch6 contient bonjour
```


Conversion d'un type primitif vers String

```
int i = 10;  
String ch = String.valueOf(i);
```

Des surcharges de la méthode `valueOf()` sont également définies pour des paramètres de type boolean, long, float, double et char

Conversion de String vers un type primitif

```
String ch = "2019";  
int n = Integer.parseInt(ch); // n vaut 2019
```

D'une manière générale on dispose des méthodes suivantes:

- `Byte.parseByte`
- `Integer.parseInt`
- `Long.parseLong`
- `Float.parseFloat`
- `Double.parseDouble`

Conversion entre chaine et tableau

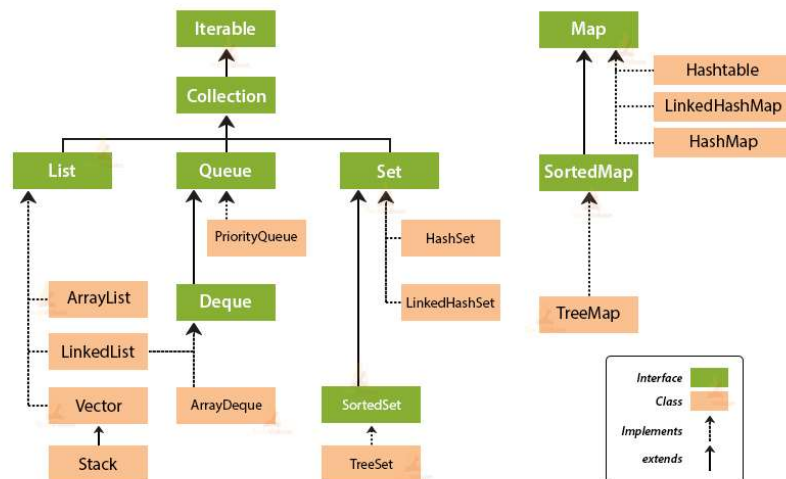
```
char tab[] = {'h','e','l','l','o'};  
String ch = new String(tab); // ch contient "hello"
```

```
// inversement  
String ch = "hello";  
char tab[] = ch.toCharArray();
```

Collections

- L'API Collections propose un ensemble d'interfaces et de classes dont le but est de stocker de multiples objets.
- Elle propose 4 grandes familles de collections, chacune définie par une interface de base
 - **List** : collection d'éléments ordonnés qui accepte les doublons
 - **Set** : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
 - **Map** : collection sous la forme d'une association de paires clé/valeur
 - **Queue** et **Deque** : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement

Collections



Classe Vector

- La classe `Vector` (package `java.util`), permet de stocker des objets dans un tableau dont la taille évolue.

Instanciation d'un vecteur

```
Vector v1 = new Vector();
```

```
Vector v2 = new Vector(taille); // prévoir une taille initiale
```

Classe Vector

Ajouter un objet

```
Point p1 = new Point(2, 3);  
v.addElement(p1);
```

- Le premier élément ajouté a l'indice 0, le suivant l'indice 1, etc...
- La méthode `size()` renvoie le nombre d'éléments contenus dans le vecteur

Lire un objet

```
Object o = v.elementAt(i);  
Point p = (Point) v.elementAt(i);
```

Classe Vector

Autres méthodes

- `contains(Object)` : indique si l'objet est contenu dans le Vector.
- `copyInto(Object[])` : copie les éléments dans un tableau classique.
- `firstElement()` : renvoie le premier élément.
- `indexOf(Object)` : renvoie l'indice de l'objet
- `insertElementAt(Object, int)` : insère l'objet à l'indice indiqué
- `isEmpty()` : indique si le Vector est vide
- `lastElement()` : renvoie le dernier élément
- `removeAllElements()` : vide le Vector
- `removeElementAt(int)` : retire l'objet dont l'indice est donné
- `setElementAt(Object, int)` : place l'objet à l'indice donné
- `size()` : renvoie le nombre d'éléments

Entrées Clavier : classe Scanner

- La lecture des entrées de clavier en Java se fait à l'aide d'un objet **Scanner** (package `java.util`)

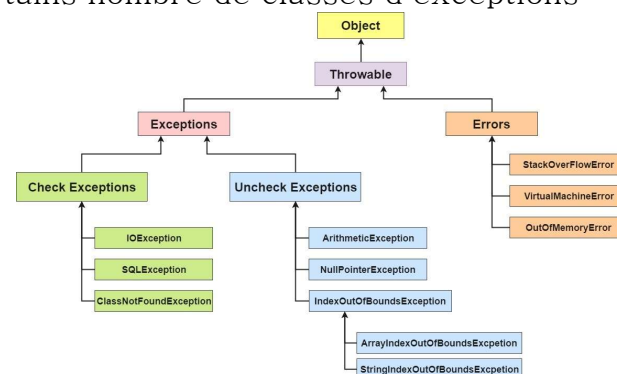
```
Scanner sc = new Scanner(System.in);
```

- La lecture se fait à l'aide d'une méthode, selon le type de l'entrée attendu

```
int x = sc.nextInt();  
char c = sc.nextChar();  
float r = sc.nextFloat();  
long n = sc.nextLong();  
String ch = sc.nextLine();
```

Exceptions

- Une exception est souvent associée à une erreur qui survient durant l'exécution d'un programme
 - Les différents types d'exceptions (d'erreurs) sont représentées via un certain nombre de classes d'exceptions



Exceptions

- Les classes **Error** correspondent aux erreurs graves de JVM
 - Il est difficile de les corriger (`java.lang.OutOfMemoryError`)
 - Elles sont relayées à la méthode appelante par défaut
- Les classes **"Check Exceptions"**
 - Obligation de dire dans le code ce que vous faites si elles sont déclenchées, sinon le programme ne compile pas
 - Soit vous capturez et traitez l'exception via l'instruction `try/catch`
 - Soit vous relayer explicitement tout déclenchement d'une telle exception à la méthode appelante
- Les classes **"Unchecked Exceptions" (Runtime Exception)**
 - Le déclenchement de ces exceptions n'est pas vérifié à la compilation
 - Elles peuvent intervenir assez fréquemment mais vous n'avez pas d'obligation de les surveiller via le couple d'instructions `try/catch`
 - Elles sont relayées à la méthode appelante

Traitement des exceptions

- Déclenchement d'une exception
 - Pour déclencher une exception, on utilise le mot clé **throw**

```
public void setDenominator (int denominator) {  
    if(denominator == 0) {  
        throw new RuntimeException("denominator cannot be 0");  
    }  
    this.denominator = denominator;  
}
```
 - Le fait de déclencher une exception de type `RuntimeException` fait que par défaut qu'elle est relayée à la méthode appelante
 - Il s'agit d'une « Unchecked Exception ». Son éventuel déclenchement n'est donc pas vérifié à la compilation

Traitement des exceptions

■ Déclenchement d'une exception

- Pour imposer aux développeurs de dire ce qu'ils doivent faire en cas de déclenchement d'exceptions, il faut alors préférer des « Checked Exceptions »

- Cela est réalisé en ajoutant le mot clé `throws` à la signature de la méthode qui déclenche potentiellement l'exception.
- En cas de levée d'exception, elle sera remontée à la méthode appelante.

```
public void setDenominator (int denominator) throws Exception {  
    if(denominator == 0) {  
        throw new RuntimeException("denominator cannot be 0");  
    }  
    this.denominator = denominator;  
}
```

Traitement des exceptions

■ Interception d'une exception

- Pour intercepter et traiter une exception, vous pouvez utiliser l'instruction « `try/catch` » ou « `try/catch/finally` »

- Le bloc « `try` » correspond au code à exécuter et pour lequel réaliser une surveillance sur un potentiel déclenchement d'exception
- Le(s) bloc(s) « `catch` » correspond(ent) au traitement à effectuer en cas de déclenchement d'une exception donnée
 - Il peut y avoir autant de bloc `catch` que souhaité.
 - Il faut, dans ce cas, les organiser en commençant avec les exceptions spécifiques et allant vers les exceptions les plus générales.
- Le bloc « `finally` » est exécuté quelque soit l'issue du bloc « `try` » qu'on soit en succès ou échec.
 - Il permet de finaliser les choses si nécessaire (fermeture de fichier, fermeture de connexion à une BD, ...)

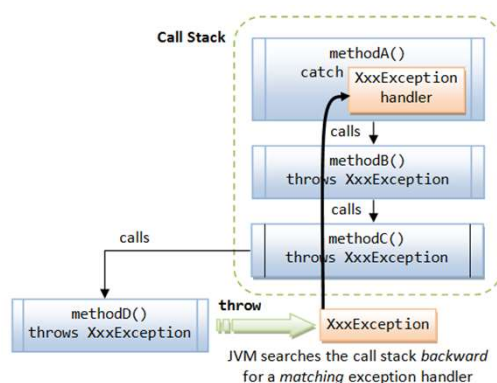
Traitement des exceptions

■ Interception d'une exception

- Syntaxe générale d'un bloc « try/catch/finally »

```
try {  
    // Instructions sous surveillance  
} catch (Exception e) {  
    // Instructions à faire si une exception se déclenche  
} finally {  
    // Instructions de finalisation  
}
```

Traitement des exceptions



```
public void methodD() throws XxxException {  
    // XxxException occurs  
    // construct an XxxException object and throw  
    // to JVM  
    if ( ... ) throw new XxxException(...);  
}
```

```
public void methodC() { // no exception declared  
    .....  
    try { .....  
        // uses methodD() which declares XxxException  
        methodD();  
        .....  
    } catch (XxxException ex) { // Exception handler for  
        XxxException  
        .....  
    }  
    finally { // optional // These codes always run, used  
        for cleaning up ..... }  
    ..... }
```

```
public void methodC() throws XxxException {  
    // uses methodD() which declares "throws XxxException"  
    methodD();  
    // no need for try-catch  
}
```