Module INFO1

Introduction à la programmation orientée objet

1ère année
ENSMM

2018

Contenu et objectifs du module

Compétences visées à l'issu du module

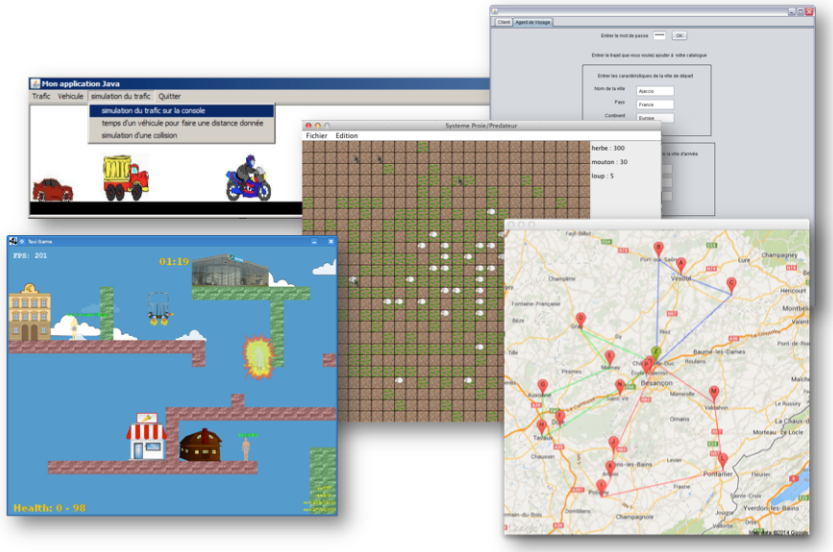
1. Programmation : utiliser les objets, définir une classe, utiliser des structures de données séquentielles et écrire les algorithmes de traitement associés, structurer des données du type liste/élément, lire et écrire des données dans des fichiers textes, créer une interface graphique élémentaire avec Swing.
2. Gestion de projet : analyser un problème, concevoir les algorithmes et les structures de données permettant sa résolution, coder proprement un logiciel, tester un logiciel, rendre compte de son travail à l'oral ou à l'écrit.

- Prérequis (programme des classes préparatoires)
  - Algorithmique
  - Complexité algorithmique
  - Sous-programmes (fonctions, procédures)
  - Récursivité

Organisation du module

Notions de base :

- Objets (2h TD + 2h TP)
- Classes (4h TD + 4h TP)
- Tableaux et algorithmes (4h TD + 4h TP)
- Listes (4h TD + 4h TP)
- Fichiers textes (4h TDAO)



Mini-projet :

- 6h TD en relation avec le projet
- 14h TP
- 4h TDAO interfaces graphiques

Évaluation

- 1 note de TP
- 1 contrôle de connaissance à mi-parcours (sur table)
- 1 mini-projet en binôme

Plan du cours

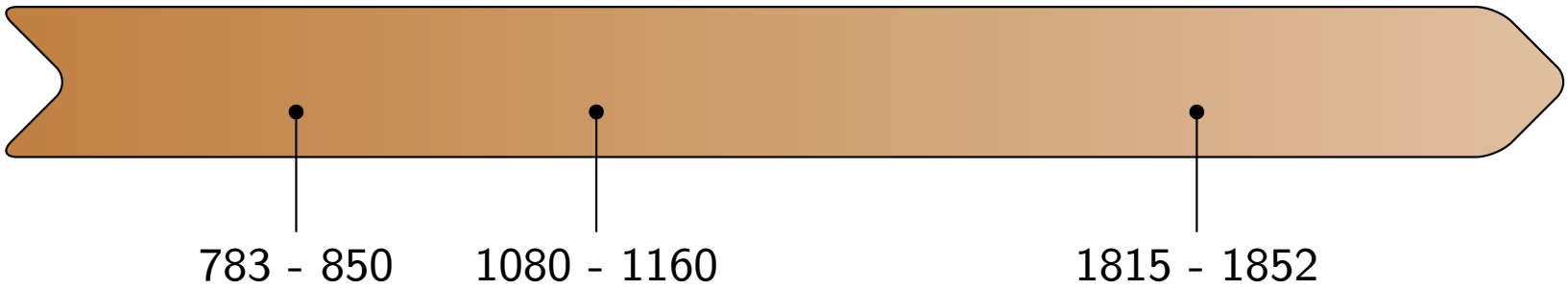
1. D'où vient Java ?
2. Comment utiliser les variables et les types de base en Java ?
3. Comment écrire un algorithme en Java ?
4. Quels sont les objectifs de la programmation orientée objet ?
5. Comment écrire une classe en Java ?

D'où vient Java ?

- Formalisation du concept d'algorithme
- Premiers langages machines
- Premiers langages symboliques
- Langages structurés de haut niveau
- Langages orientés objets

Formalisation du concept d'algorithme

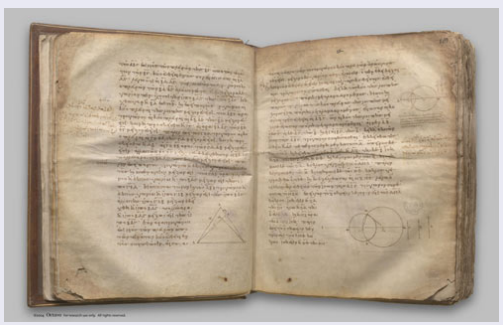
Naissance du concept d'algorithme



Muhammad Al Khawarizmi

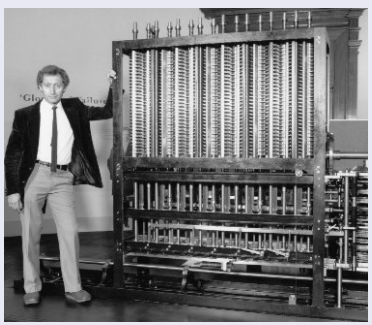
Mathématicien perse, auteur d'un ouvrage qui décrit des méthodes systématiques de calculs algébriques

Adelard de Bath



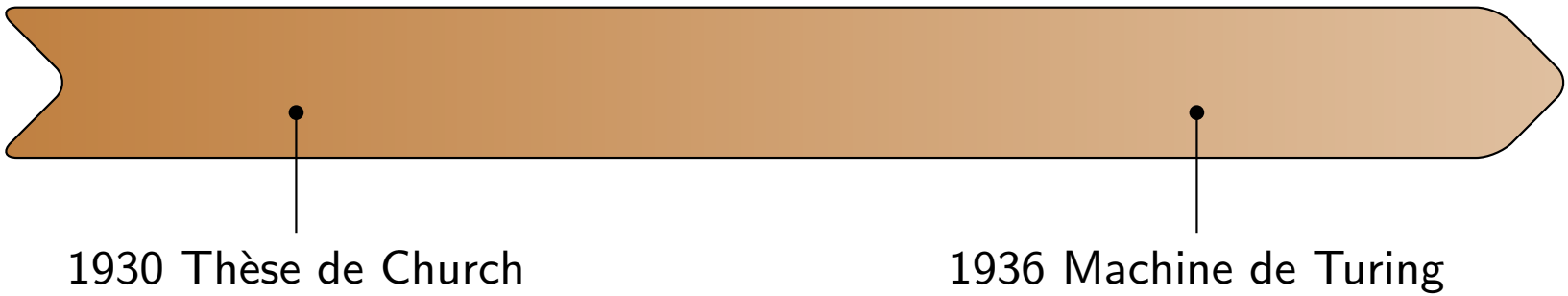
Mathématicien britannique, introduit le terme latin de *algorismus*. Ce mot donnera **algorithme** en français en 1554

Ada Lovelace

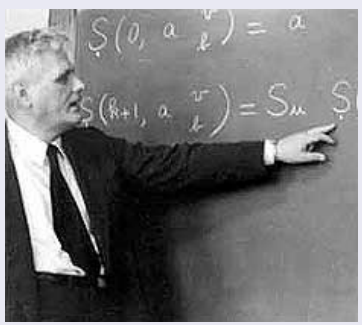


Mathématicienne britannique, propose une méthode très détaillée pour calculer les nombres de Bernoulli considérée comme le premier programme informatique pour la machine à différence de Charles Babbage

Formalisation du concept d'algorithme

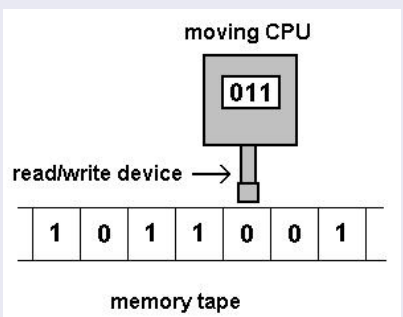


Alonzo Church (1903-1995)



Mathématicien américain, formalise les concepts d'algorithme, du λ-calcul et de décidabilité. Ses travaux influencèrent les langages de programmation fonctionnelle.

Alan Turing (1912-1954)

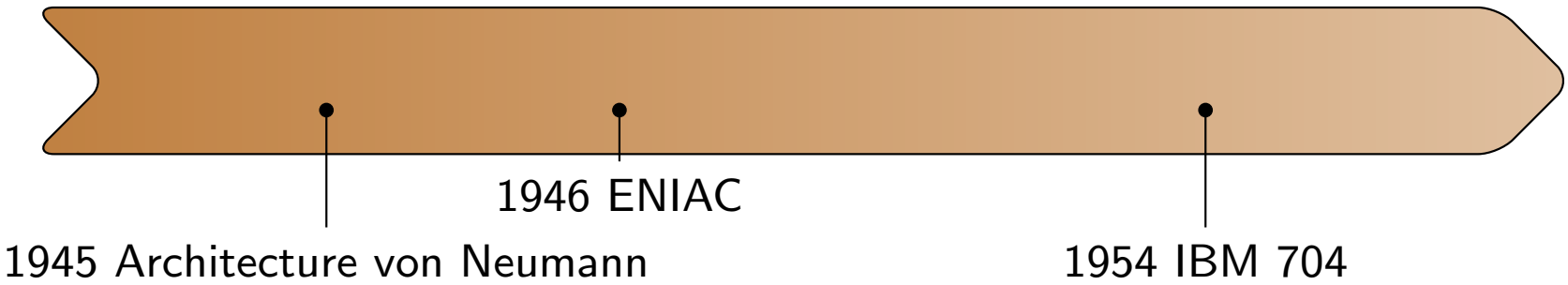


Mathématicien britannique, propose un modèle abstrait : la machine de Turing. Il montre notamment que :

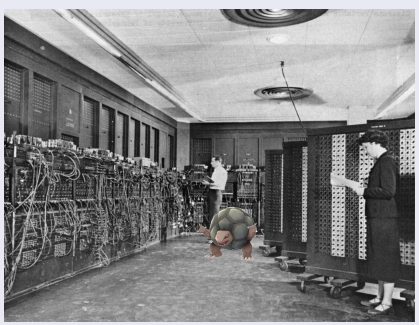
- Tout problème calculable peut être calculé par une machine de Turing
- Le problème de l'arrêt est indécidable

Premiers langages machines

Premiers langages machines



ENIAC



Premier ordinateur électronique (à tubes) complet au sens de Turing.

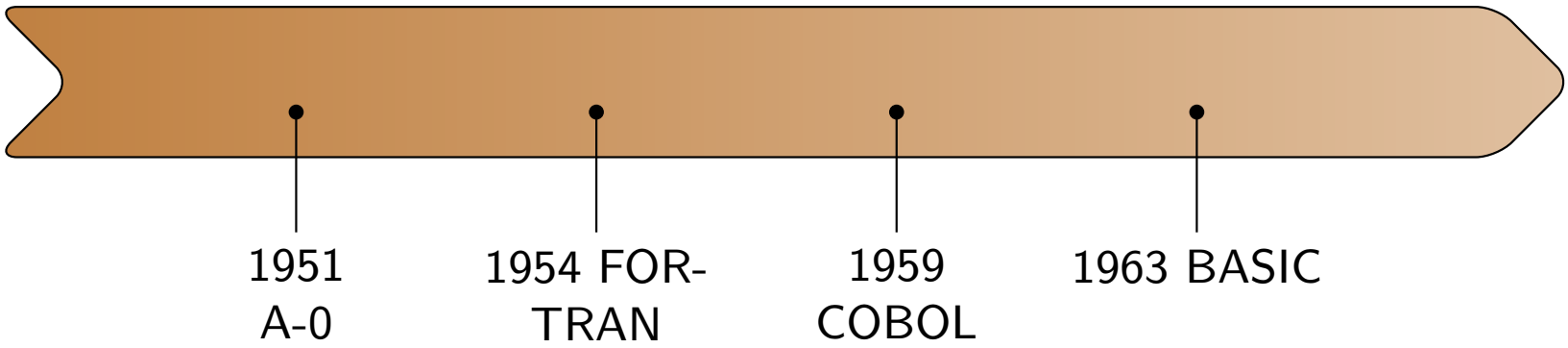
IBM 704

Adresse	Contenu de la mémoire	
00000000	01010101	1ère instruction
00000001	11111111	2ème instruction
00000010	00000000	
00000011	11001100	
00000100	00110011	début de la 3ème instruction

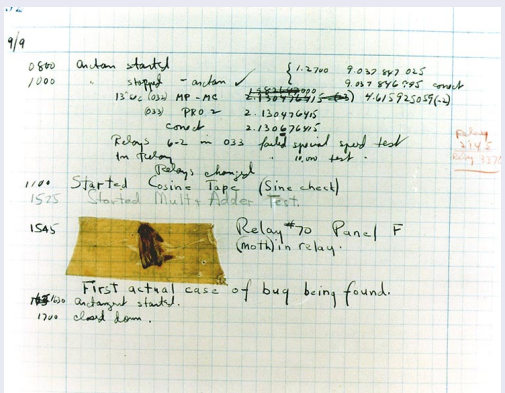
Programmes écrits avec les instructions de base de la machine (langage machine)

Premiers langages symboliques

Premiers langages symboliques

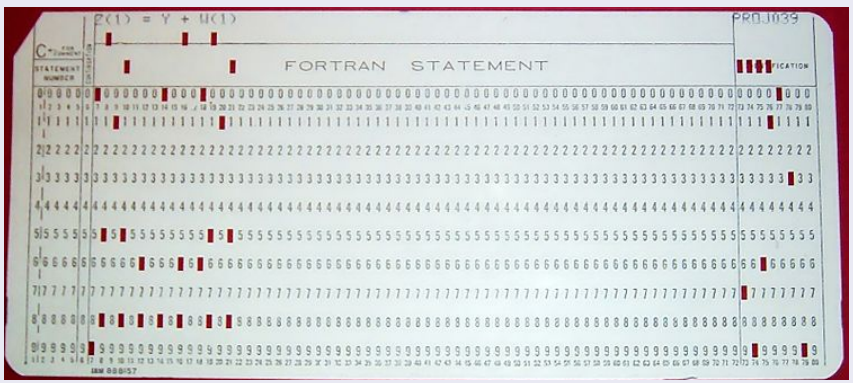


Grace Murray Hopper (1906-1992)



Informaticienne américaine, conceptrice du premier compilateur (A-0 System) et à l'origine du langage COBOL.

John Backus (1924-2007)



Informaticien chez IBM, publie en 1954 un article titré *Specifications for the IBM Mathematical FORmula TRANslating System*

Premiers langages symboliques

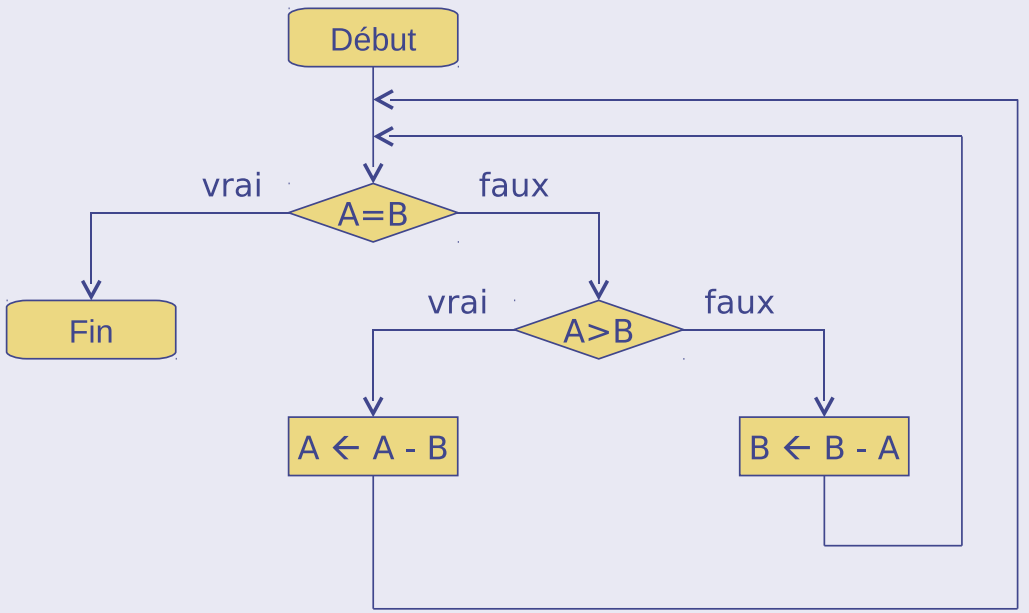
Kemeny & Kurtz (1963)

Conçoivent le BASIC au Dartmouth College

BASIC

```
10 if A = B goto 70
20 if A < B goto 50
30 A = A - B
40 goto 10
50 B = B - A
60 goto 10
70 end
```

Organigramme (IBM 1969, norme ISO 5807 en 1985)



- Programmes difficiles à vérifier et à réutiliser
- Programmation ≪ spaghetti ≫
- Organigrammes limités à des algorithmes très simples

Langages structurés de haut niveau

Langages structurés de haut niveau

Années 70

Nombreuses publications sur le génie logiciel

- 1966 : Böhm et Jiacopini, *Go To Statement Considered Harmful*
- 1970 : Warnier, *Principes de la Logique de Construction de Programmes*
- 1975 : Dijkstra, *Guarded commands, non determinacy and formal derivation of programs*

Introduction de nouveaux concepts

- Structures de contrôles (tant que)
- Structures de données (enregistrements)

Dennis Ritchie (1941-2011)

Invente le langage C pour réécrire UNIX aux Bell labs en 1972

GnuTLS : un GoTo responsable d'une faille critique

Qui existerait depuis 2005 dans la bibliothèque très utilisée sous Linux

Le 7 mars 2014, par Arsene Newman, Chroniqueur Actualités

r /> Edsger Dijkstra, informaticien de renom, nous avait déjà prévenu en 1968 que les sauts inconditionnels de type GoTo pouvaient s'avérer dangereux ; il avait même publié un article sur le sujet, intitulé GoTo Statement Considered Harmful.

46 ans plus tard, la chasse aux bugs et aux failles informatiques le prouve encore. En effet GnuTLS l'une des bibliothèques les plus populaires pour la gestion des certificats électroniques et l'implémentation des protocoles SSL & TLS sous Linux, serait affectée par une faille majeure due à un mauvais GoTo. Vu la popularité de la bibliothèque, on estime que des centaines d'outils pourraient être affectés à leur tour. C'est le cas par exemple de l'utilitaire apt-get utilisé sous Ubuntu et Debian ainsi que le client http en ligne de commande lib-curl dans sa version 3.

Dénommée CVE-2014-0092, la faille a été découverte par Nikos Mavrogiannopoulos, membre de l'équipe des technologies de sécurité de RedHat, lors d'un audit de GnuTLS. Elle existerait depuis 2005.

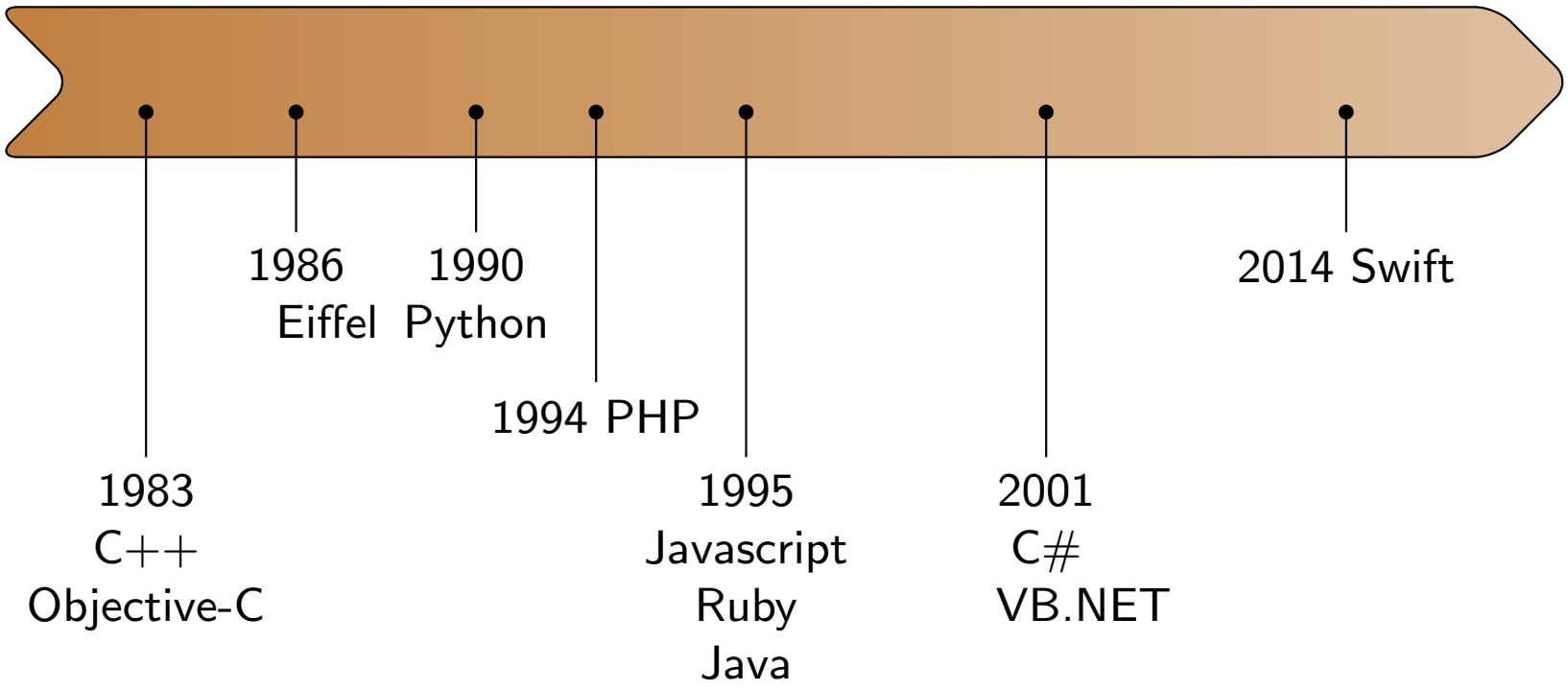
La faille serait due à une erreur de branchement GoTo (*goto cleanup* au lieu d'un *goto fail*) dans une section du fichier verify.c, responsable de la vérification de l'authenticité des certificats x509, ce qui aurait pour conséquence une terminaison inhabituelle du code de vérification et une détection erronée de l'authenticité d'un certificat électronique. Selon RedHat, un attaquant pourrait profiter de la faille pour créer un certificat « *qui serait accepté par GnuTLS comme valide pour un site choisi par l'attaquant* ».

Tout cela explique l'empressement de RedHat. Elle recommande le passage à la version 3.2.12 ou le fix des versions 2.12.x, grâce à un patch publié dans la foulée.

Enfin cette faille vient nous rappeler celle découverte sous les systèmes d'exploitation Apple, à savoir le fameux *goto fail* affectant les implémentations des protocoles SSL et TLS [lire l'article de la rédaction], même si pour beaucoup d'experts le *goto cleanup* de Linux serait bien pire et plus critique que le *goto fail* d'Apple.

Langages orientés objets

Langages orientés objets

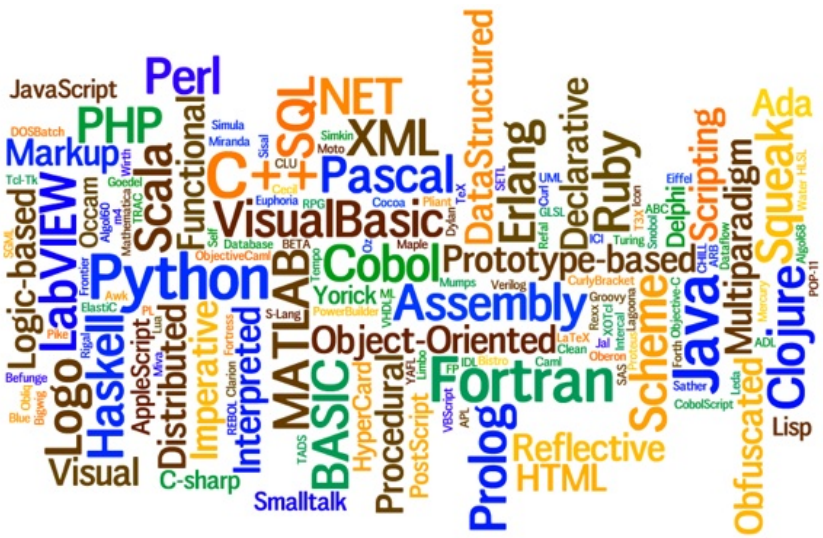


Introduction de nouveaux concepts

- Objets
- Encapsulation
- Héritage
- Polymorphisme

Situation actuelle

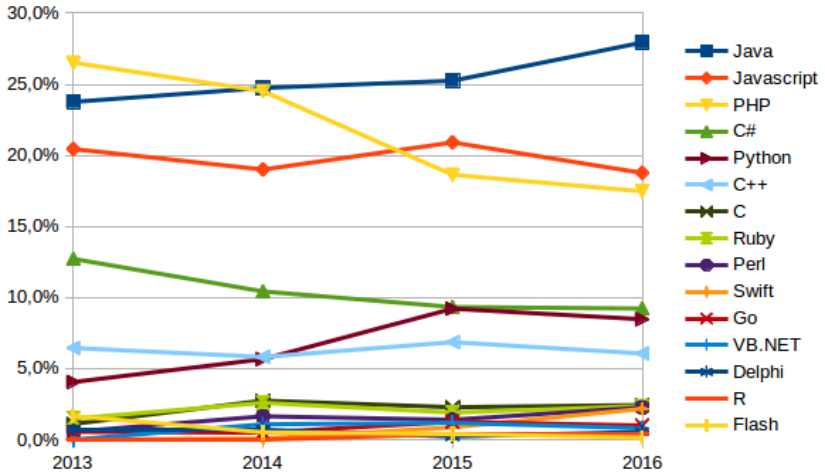
- Des centaines de langages
- Langages déclaratifs
  - Programmation fonctionnelle
  - Programmation logique
- Langages impératifs
  - Programmation procédurale
  - **Programmation orientée objet**



Situation actuelle

Language Rank	Types	Jobs Ranking
1. Java		100.0
2. C		99.4
3. Python		99.3
4. C++		92.2
5. JavaScript		89.9
6. C#		86.4
7. PHP		80.5
8. HTML		79.7
9. Ruby		76.6
10. Swift		76.4

IEEE Top Programming Languages 2017 : Focus on Jobs Demands.



Offres d'emploi postées sur le portail Emploi de Developpez.com (plus de 15 000 offres en juin 2016).

Remarque

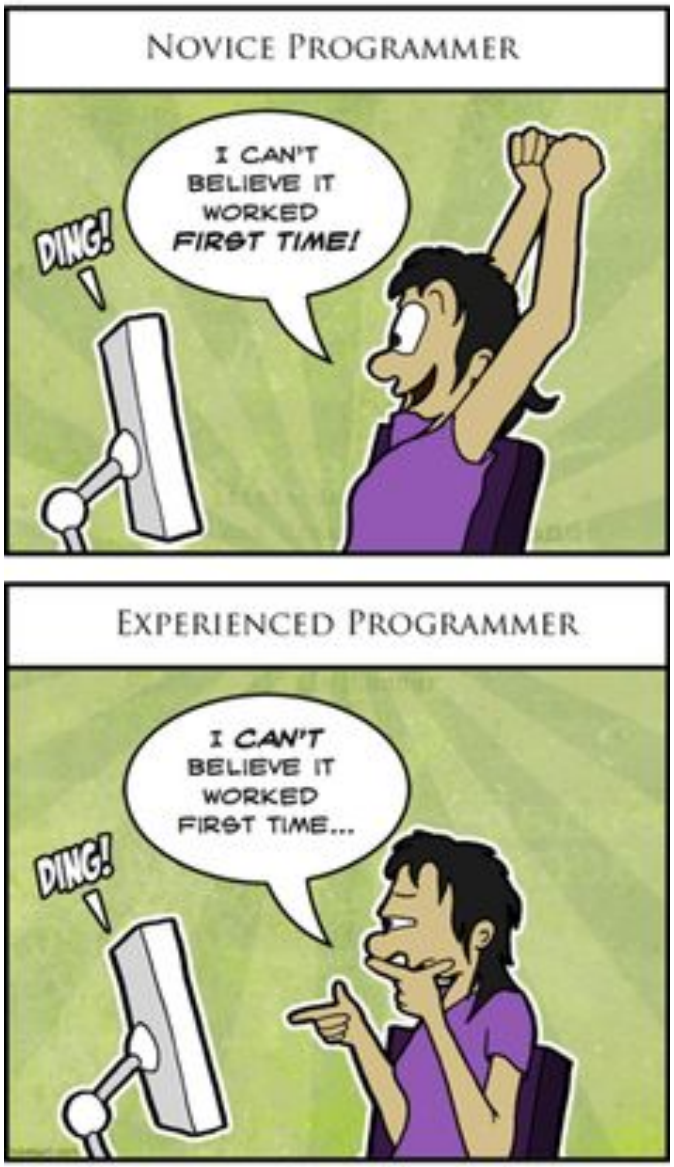
La POO (qui regroupe en autres Java, C++, C#, Objective-C, Python) est de loin la technique la plus répandue (plus de 75% des développements)

Conclusion

- La POO est le paradigme de programmation le plus répandu (et notamment Java)
- Les langages impératifs sont complets au sens de Turing
- L'arrêt d'un programme impératif n'est pas décidable

Bonnes pratiques

1. Avant l'écriture d'un programme : analyser et structurer
2. Pendant l'écriture d'un programme : coder proprement !
3. Après l'écriture d'un programme : tester, retester, reretester, ...



Comment utiliser les variables et les types de base en Java ?

- Identificateurs et variables
- Types primitifs
- Chaînes de caractères
- Bonnes pratiques

Identificateurs et variables

Qu'est-ce qu'un identificateur ?

Définition

Un identificateur est le nom associé à une variable, à une classe, ou à une méthode. En Java, un identificateur est une suite de caractères commençant par une lettre et ne contenant pas d'espace.

- `unEntier`, `x`, `x2`, `Point`, `MAX_VALUE` sont des identificateurs valides
- 3a, `couleur fond` ne le sont pas

Remarque

`Xi` est un identificateur, mais `i` n'est pas un indice

Comment déclarer une variable ?

Typage

Java est un langage ayant un *typage statique fort* (contrairement à Python)

- Une variable a un type unique et connu
- Une variable doit être déclarée avant d'être utilisée
- Une variable ne peut pas changer de type (typage statique)
- Pas de mélange entre types : une variable d'un type ne peut recevoir que des valeurs ayant le même type (typage fort)

Exemple 1

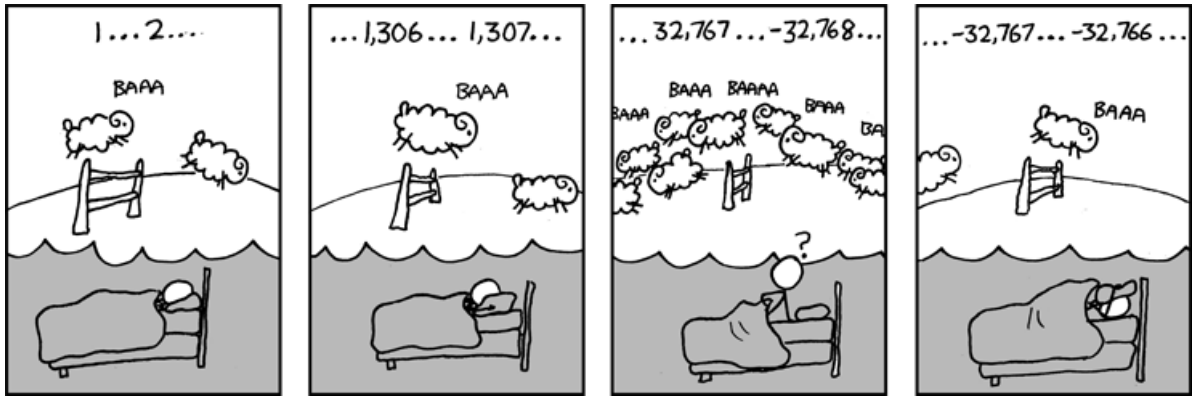
```
int x,y,z;

x = 5;
y = 13 + (56 * 3);
z = x + y;
```

Exemple 2

```
int x;
char z;

x = 5;
z = x; // erreur !
```

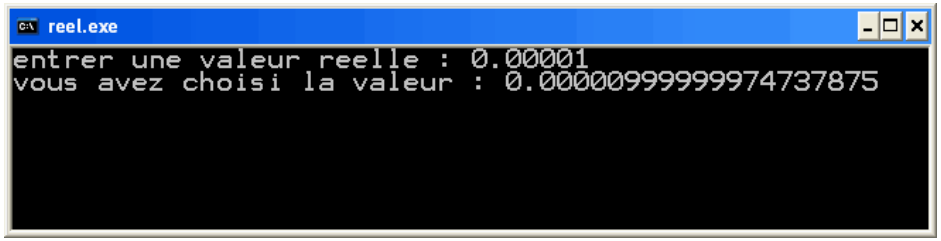
Types primitifs

Types entiers



- Entiers signés
  - `byte` : entier sur 1 octet (-128 à 127)
  - `short` : entier sur 2 octets (-32 768 à 32 767)
  - `int` : entier sur 4 octets (-2 147 483 648 à 2 147 483 647)
  - `long` : entier sur 8 octets (-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807)
- Opérateurs de calcul : `+`, `-`, `*`, `/`, `%`
- Opérateurs relationnels : `<`, `>`, `<=`, `>=`, `==`, `!=`
- Méthodes : `Math.abs(int x)`, `Math.max(int x, int y)`, `Math.min(int x, int y)`, etc.

Types réels



- Nombre à virgule flottante de la forme $signe \times mantisse \times 2^{exp}$
  - `float` : réel sur 4 octets (1,4e-45 à 3,4e38, 6 chiffres significatifs)
  - `double` : réel sur 8 octets (4,9e-324 à 1,8e308, 14 chiffres significatifs)
- Opérateurs de calcul : +, -, *, /
- Opérateurs relationnels : <, >, <=, >=, (ne pas utiliser ==)
- Méthodes : `Math.abs(double x)`, `Math.max(double x, double y)`, `Math.sqrt(double x)`, `Math.cos(double a)`, `Math.pow(double x, double exp)`, `Math.exp(int x)`, `Math.round(double x)`, etc.

Remarque

On ne peut représenter que 4 294 967 296 réels avec un `float` !

Type booléen

- Type booléen : `boolean`
- Valeurs : `true`, `false`
- Opérateurs relationnels : ==, !=
- Opérateurs logiques : ! (non), && (et), || (ou)

Exemple

```
boolean a, b, c;
a = true;
b = !a;
c = (a && b);

if (c) {
    b = (10 > 3);
}
```

Type caractère

- Type caractère : `char`
- Jeu de caractères UNICODE UTF-16
- Exemples de valeurs : `'a' 'b' 'A' 'B' '0' '1' '2' '<' '*' '+' '!' '(' '['`
- Relation d'ordre : `'0'` < `'1'` < ... < `'a'` < `'b'` < ... < `'z'` < `'A'` < ... < `'Z'`

Exemple

```
char unCaractere;

unCaractere = 'a';

if (unCaractere == '1') {
    ...
}

System.out.println(unCaractere);
```

Remarque

Utilisation des guillemets simples

Chaînes de caractères

Chaînes de caractères

- Chaînes de caractères : Classe `String` (n'est pas un type primitif)
- Jeu de caractères UNICODE UTF-16

Exemple

```
String uneChaine;
uneChaine = "Bonjour Antoine";

String question = ", comment vas-tu ?";
uneChaine = uneChaine + question;

if (uneChaine.equals("bonjour antoine, comment vas-tu ?")) {
    ...
}
```

Remarques

- Utilisation des guillemets doubles
- Ne pas utiliser == pour la comparaison mais la méthode `equals()`

Bonnes pratiques

Rendre les programmes lisibles

Que fait ce programme ?

```
f = false;
while (!f) {
    r = a % b;
    if (r == 0) {
        p = b;
        f = true;
    } else {
        a = b;
        b = r;
    }
}
return p;
```

Rendre les programmes lisibles

Que fait ce programme ?

```
f = false;
while (!f) {
    r = a % b;
    if (r == 0) {
        p = b;
        f = true;
    } else {
        a = b;
        b = r;
    }
}
return p;
```

Que fait ce programme ?

```
fin = false;
while (fin == false) {
    reste = nombrePlusGrand % nombrePlusPetit;
    if (reste == 0) {
        plusGrandCommunDiviseur = nombrePlusPetit;
        fin = true;
    } else {
        nombrePlusGrand = nombrePlusPetit;
        nombrePlusPetit = reste;
    }
}
return plusGrandCommunDiviseur;
```

Comment nommer une variable ?

Règle n°1

L'identificateur d'une variable commence par une minuscule et utilise le *camelCase*

Règle n°2

L'identificateur d'une variable doit être compréhensible et prononçable

Règle n°3

L'identificateur d'une variable doit donner du sens au programme et rendre inutile tout commentaire

Remarque

On utilisera des identificateurs courts uniquement dans les cas suivants :

- `i`, `j`, `k` pour les indices et compteurs
- `x`, `y`, `z` pour des coordonnées

Comment écrire un algorithme en Java ?

- Affectation
- Séquence
- Structures de choix
- Structures d'itération
- Bonnes pratiques

Affectation

Affectation

Pseudo-langage

```
x ← 5
y ← 7,6 + x
c ← 't'
present ← VRAI
test ← (5 < 10)
```

Java

```
x = 5;
y = 7.6 + x;
c = 't';
present = true;
test = (5 < 10);
```

- Affecte une valeur à une variable

- Déroulement :
  1. Évaluation de l'expression (à droite)
  2. Mise en mémoire du résultat dans la variable (à gauche)

Séquence

Séquence

- Une séquence est délimitée par des accolades (bloc)
- Les instructions d'un bloc s'exécutent dans l'ordre de leur écriture (de gauche à droite et de haut en bas)
- L'existence d'une variable est limitée au bloc où elle est déclarée

```
{
    int i = 4;
    i = i + 1;
    {
        double x;
        x = Math.PI * i;
    }
    // x n'existe pas ici
}
```

Séquence

- Une séquence est délimitée par des accolades (bloc)
- Les instructions d'un bloc s'exécutent dans l'ordre de leur écriture (de gauche à droite et de haut en bas)
- L'existence d'une variable est limitée au bloc où elle est déclarée

```
{
    int i = 4;
    i = i + 1;
    {
        double x;
        x = Math.PI * i;
    }
    // x n'existe pas ici
}
```

Bonnes pratiques

1. Indenter le code
2. Une seule instruction par ligne

Structures de choix

Structure de choix *si-alors-sinon*

Pseudo-langage

```
si ( condition ) alors
    séquence 1
sinon
    séquence 2
finsi
```

Java

```
if ( condition ) {
    // séquence 1
} else {
    // séquence 2
}
```

- La condition doit être une expression booléenne
- Si la valeur de la condition est *vraie* alors la séquence 1 est exécutée
- Si la valeur de la condition est *fausse* alors la séquence 2 est exécutée
- Après quoi, l'exécution reprend à la première instruction qui suit *finsi*
- Remarque : la séquence 2 est optionnelle

Structure de choix *si-alors-sinon*

Pseudo-langage

si $(n = 10)$ alors
    $n \leftarrow 0$
sinon
    $n \leftarrow n + 1$
finsi

si $(a \geq 1$ et $b \neq 0)$ alors
    $c \leftarrow a * 3 / b$
    $a \leftarrow 0$
finsi

Java

```
if (n == 10) {
    n = 0;
} else {
    n = n + 1;
}
```

```
if (a >= 1 && b != 0) {
    c = a * 3 / b;
    a = 0;
}
```

Structure de choix *si-alors-sinon*

Pseudo-langage

```
si (n = 10) alors
    n ← 0
sinon
    n ← n + 1
finsi


si (a ≥ 1 et b ≠ 0) alors
    c ← a * 3 / b
    a ← 0
finsi
```

Java

```
if (n == 10) {
    n = 0;
} else {
    n = n + 1;
}


if (a >= 1 && b != 0) {
    c = a * 3 / b;
    a = 0;
}
```

Remarque

Attention aux pièges des opérateurs = en Java : l'affectation s'écrit `a=b;`, le test d'égalité s'écrit `a==b` pour les types primitifs et `a.equals(b)` pour les classes

Structure à choix multiples

Solution 1

```
if (x == -1) {
    a = a + 1;
} else if (x == 3) {
    a = a - 1;
} else if (x == 15 || x == 20) {
    a = 0;
    arret = true;
} else {
    arret = false;
}
```

Solution 2 (switch)

```
switch (x) {
    case -1:
        a = a + 1;
        break;
    case 3:
        a = a - 1;
        break;
    case 15:
    case 20:
        a = 0;
        arret = true;
        break;
    default:
        arret = false;
}
```

Exercice 1

```
int a = 22;
int b = -12;
int c = a + b;
a = c - 2 + Math.abs(b);
b = a * b / c;
```

Exercice 2

```
double x = 5.3e10;
double y = 178.6;
double max;
if (x < y) {
    max = y;
} else {
    max = x;
}
```

Exercice 3

```
int a = 3;
int b = 5;
if  (a < b && b == 10) {
    a = b;
} else {
    if (b == 5) {
        a = a + b;
    }
    b =  b + 1;
}
```

Exercice 4

```
int a = 3;
int b = 5;
int c;
if (a > 2 * b) {
    c = 8;
} else {
    c = 2;
}
if (c + 1 == a || b > c) {
    c = c + 3;
    if (c == b) {
        a = c;
    }
} else {
    if (c == a) {
        b = a * c;
    }
}
```

Structures d'itération

Structure d'itération *Tant que*

Pseudo-langage

```
tant que ( condition ) faire
    séquence
fintantque
```

Java

```
while ( condition ) {
    // séquence
}
```

- La condition doit être une expression booléenne
- Répète la séquence tant que la valeur de la condition est *vrai*
- La condition n'est testée qu'au début de chaque boucle

Remarques

- Toute structure *tant que* doit être précédée de l'initialisation des variables intervenant dans l'expression booléenne de la condition
- La séquence doit modifier au moins l'une des variables intervenant dans la condition. Il est indispensable que la condition prenne la valeur *faux* à un moment donné, afin de sortir de la boucle

Structure d'itération *Tant que*

Pseudo-langage

somme $\leftarrow$ 0
i $\leftarrow$ 1
tant que (i $\leq$ 10) **faire**
    somme $\leftarrow$ somme + i
    i $\leftarrow$ i + 1
fintantque

somme $\leftarrow$ 0
i $\leftarrow$ 10
tant que (i $\neq$ 0) **faire**
    somme $\leftarrow$ somme + i
    i $\leftarrow$ i $-$ 1
fintantque

Java

```java
int somme = 0;
int i = 1;
while (i <= 10) {
    somme = somme + i;
    i = i + 1;
}
```

```java
somme = 0;
i = 10;
while (i != 0) {
    somme = somme + i;
    i = i - 1;
}
```

Exercice 5

```
int a = 2;
int b = 5;
int c = 0;
while (b >= a && c <=10) {
    b = b - 1;
    c = c + 2;
}
```

Exercice 6

```
int a = 4;
int b = 6;
while (a <= (b / 2)) {
    a = a + 1;
}
```

Exercice 7

```
int a = 2;
int b = 0;
while (a < 10) {
     b = b + 1;
}
```

Exercice 8

```
int i = 2;
int n = 35;
boolean trouve = false;
while (i <= Math.sqrt(n) && trouve == false) {
    if ( (n % i) == 0) {
        trouve = true;
    } else {
        i = i + 1;
    }
}
```

Structure d'itération *Répéter*

Pseudo-langage

```
répéter
    séquence
tant que ( condition )
```

Java

```
do {
    // séquence
} while ( condition );
```

- Répète la séquence tant que la valeur de la condition est *vrai*
- La condition n'est testée qu'à la fin de chaque boucle

Remarque

- La séquence est exécutée au moins une fois

Exercice 9

```
int somme = 0;
int x = 0;
do {
    somme = somme + x;
    x = x - 1;
} while (x >= 1);
```

Exercice 10

```
int somme = 0;
int x = 5;
do {
    somme = somme + x;
    x = x - 1;
} while (x >= 1);
```

Structure d'itération *Pour*

Pseudo-langage

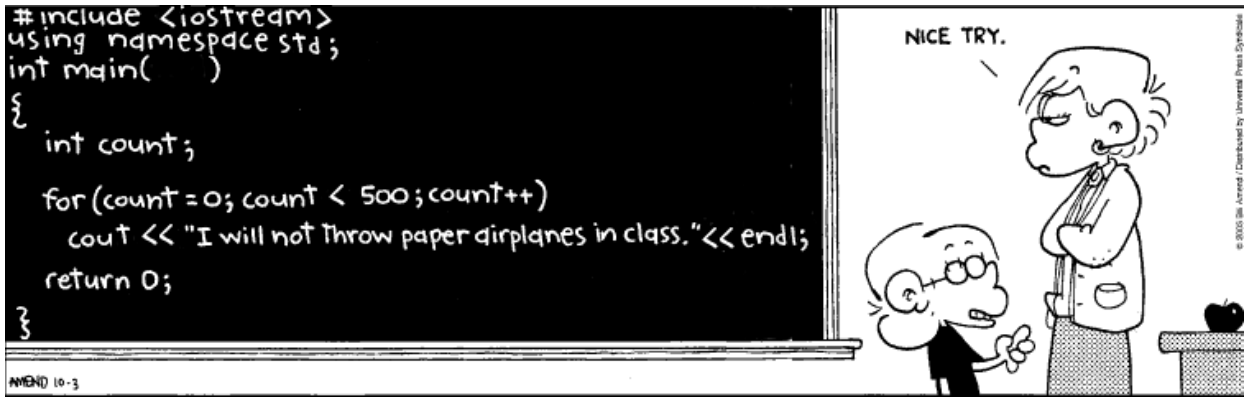
```
pour var allant de min à max (par pas de incr) faire
    séquence
finpour
```

Java

```
for (type var = min; var <= max; var = var + incr) {
    // séquence
}
```

- La séquence est exécutée pour les valeurs de *var* successivement égale à *min*, *min+incr*, *min+2*incr*, ... , *max*
- Le nombre d'itérations est connu à l'avance
- *incr* peut être négatif, il faut adapter les bornes et le test en conséquence

Structure d'itération *Pour*



Pseudo-langage

somme ← 0
pour i **de** 1 **à** 10 **faire**
    somme ← somme + i
finpour

somme ← 0
pour i **de** 20 **à** 2 **par pas de** −2 **faire**
    somme ← somme + i / 2
finpour

Java

```
int somme = 0;
for (int i = 1; i <= 10; i = i + 1) {
    somme = somme + i;
}

int somme = 0;
for (int i = 20; i >= 2; i = i - 2) {
    somme = somme + i / 2;
}
```

Structure d'itération *Pour chaque*

Pseudo-langage

```
pour chaque élément var de la collection faire
    séquence
finpour
```

Java

```
for (ClasseElement var: collection) {
    // séquence
}
```

- La séquence est exécutée pour les valeurs de *var* successivement égales au premier élément de la collection, au second, ... , jusqu'au dernier

Exercice 11

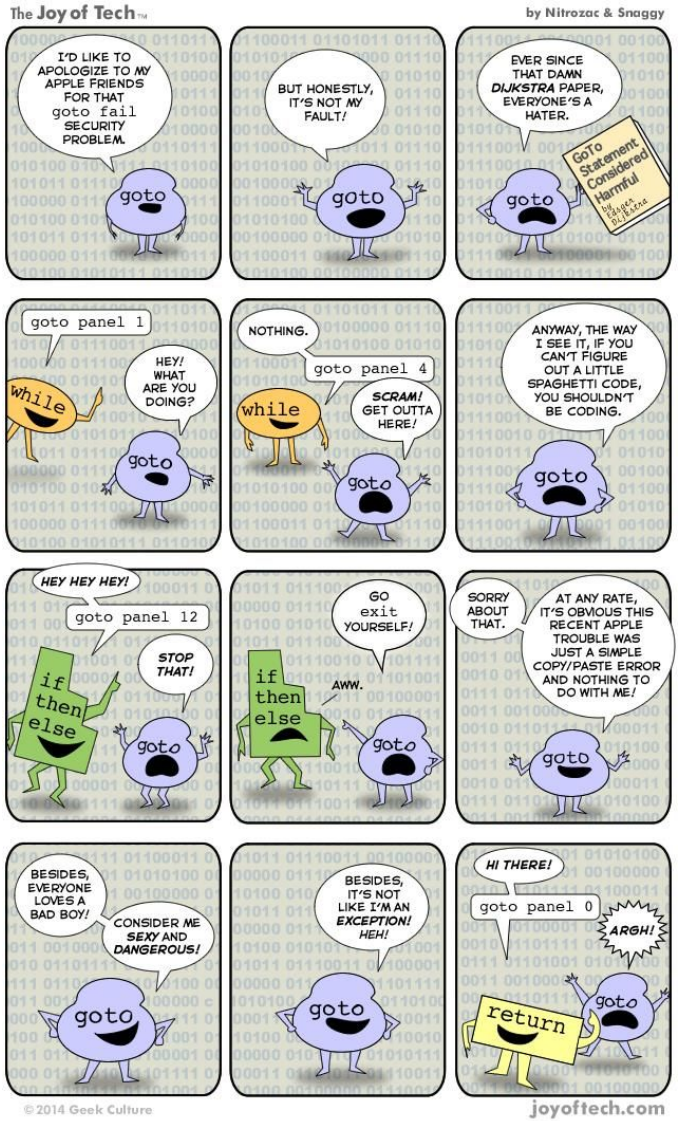
```
int s = 0;
for (int i = 1; i <= 5; i = i + 1) {
    s = s + i * i;
}
```

Exercice 12

```
int n = 0;
int max = 10;
for (int i = 1; i <= max; i = i + 1) {
    if (i * i < max) {
        n = n + 1;
    }
}
```

Bonnes pratiques

Bonnes pratiques



Règle n°1

Il est interdit de modifier la variable de contrôle ou les bornes dans une boucle `for` (si besoin utiliser une boucle `while`)

Règle n°2

Une boucle ne doit avoir qu'un point de sortie (`break` et `return` interdits)

Règle n°3

Éviter les boucles imbriquées en structurant mieux votre code (ajout de méthodes ou de classes)

Quels sont les objectifs de la programmation orientée objet ?

Encapsulation des données

- Cacher aux autres programmeurs les détails de l'implantation
- Sécuriser les données
- Délimiter la portée des évolutions futures

Maintenir la cohérence entre les données et les algorithmes

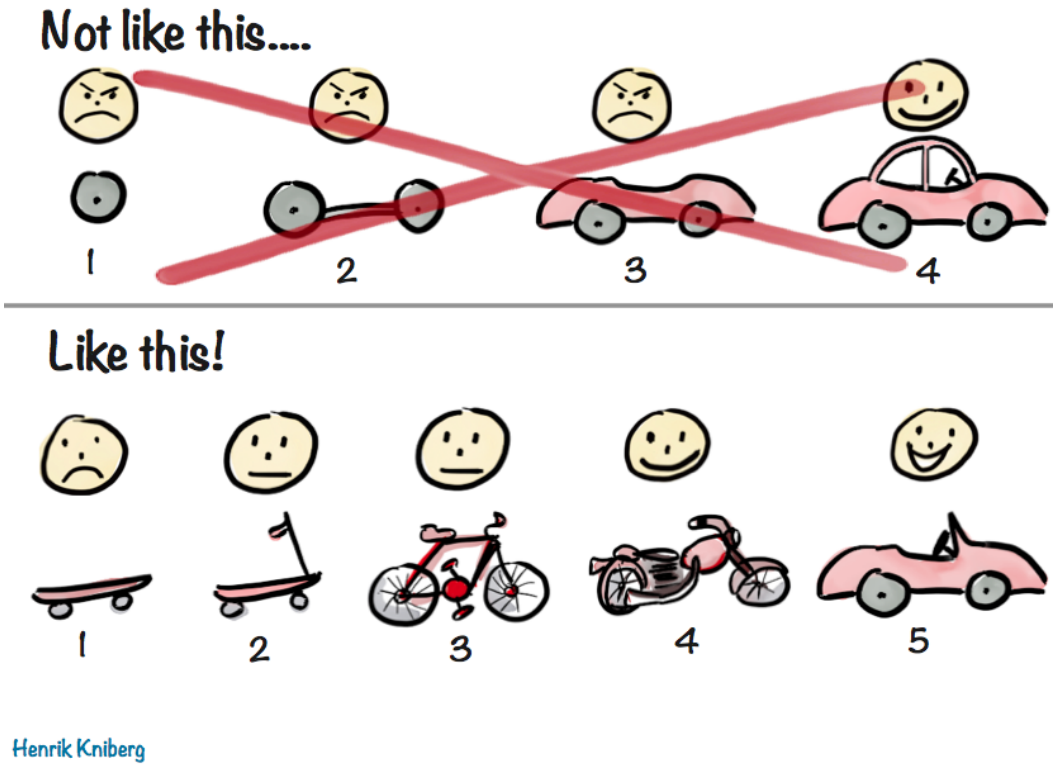
```
def norme(a):
    sommeDesCarres = a[0]**2 + a[1]**2 + a[2]**2
    return sqrt(sommeDesCarres)

# mauvaise utilisation
u = norme([1, 2])
```

Empêcher les effets de bords

```
def norme(a):
    sommeDesCarres = a[0]**2 + a[1]**2 + a[2]**2
    a[0] = 4
    return sqrt(sommeDesCarres)
```

Modularité

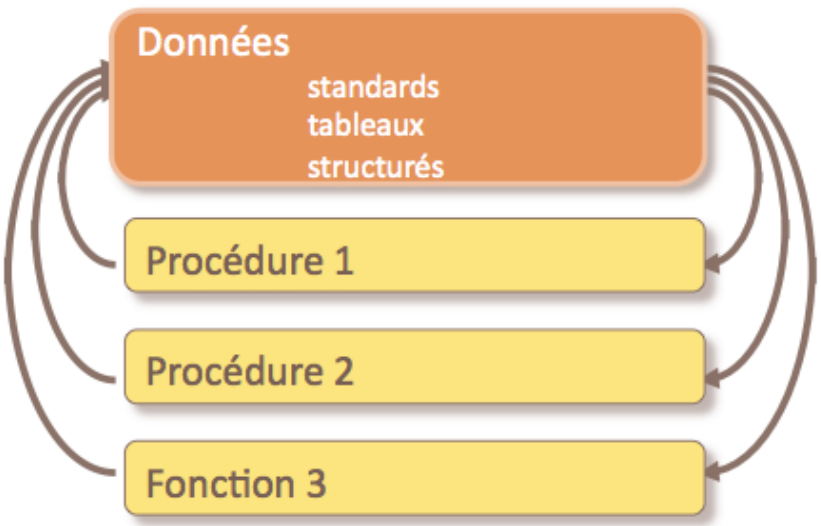
- Simplifier la conception
- Interchanger facilement des composants
- Réutiliser les modules déjà programmés et testés
- Ajouter facilement de nouvelles fonctionnalités (processus itératif)



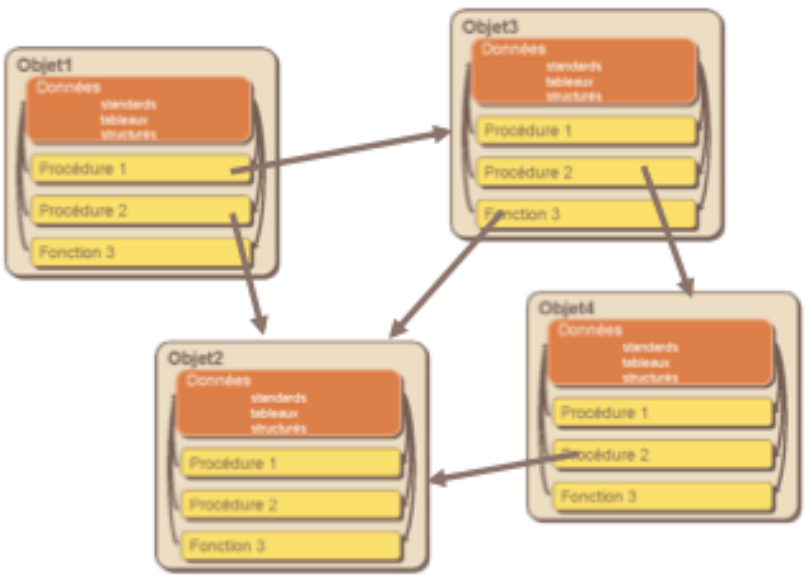
Qu'est ce que la programmation orientée objet ?

Définition

La programmation orientée objets est une méthode de mise en œuvre dans laquelle les programmes sont organisés comme des ensembles de modules coopérants (objets) encapsulant leurs données (attributs) et leurs algorithmes (méthodes).



Programmation procédurale



Programmation orientée objet

Comment écrire une classe en Java ?

- Qu'est-ce qu'un objet ?
- Qu'est-ce qu'une classe ?
- Comment définir une classe en Java ?
- Comment encapsuler les données ?
- Qu'est ce qu'une référence ?
- Bonnes pratiques

Qu'est-ce qu'un objet ?

Qu'est-ce qu'un objet ?

Création et utilisation d'un objet

1. Déclaration d'une variable pour stocker la *référence* de l'objet
   `NomDeClasse nomObjet;`
2. Construction d'un nouvel objet
   `nomObjet = new NomDeClasse( arguments de construction );`
3. Accès aux membres (publics) de l'objet
   `nomObjet.methode( arguments de la méthode );`

Exemple d'utilisation de la classe Voiture

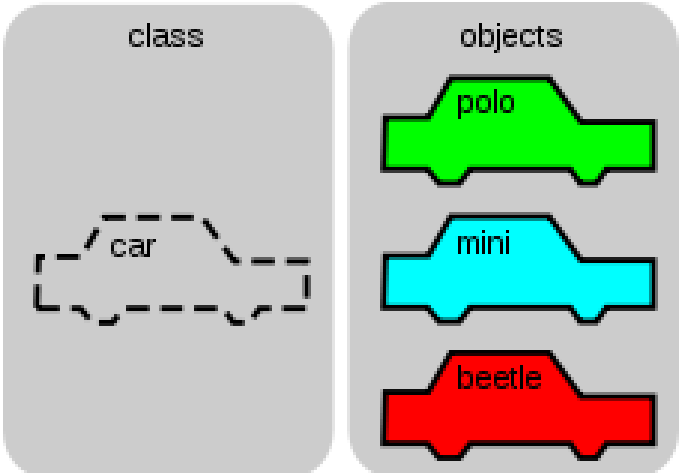
```
Voiture maTwingo;
maTwingo = new Voiture("Vert", 3, 0.0, true);
maTwingo.setCouleur("Bleu");
maTwingo.afficher() ;

Voiture monEspace = new Voiture("Noir", 5, 110, false) ;
monEspace.accelere(20) ;
System.out.println("Distance = " + monEspace.getDistanceDeSecurite()) ;
```

Qu'est-ce qu'une classe ?

Qu'est-ce qu'une classe ?

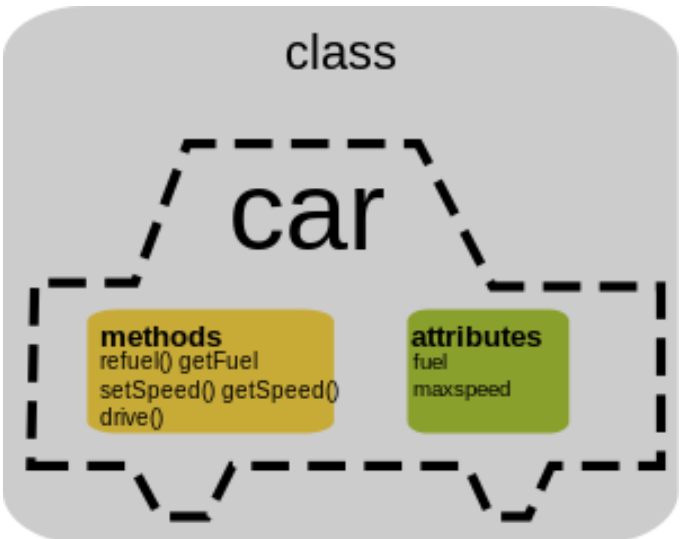
Définition

Une classe est un modèle d'objets décrivant les propriétés et comportements communs à ces objets.

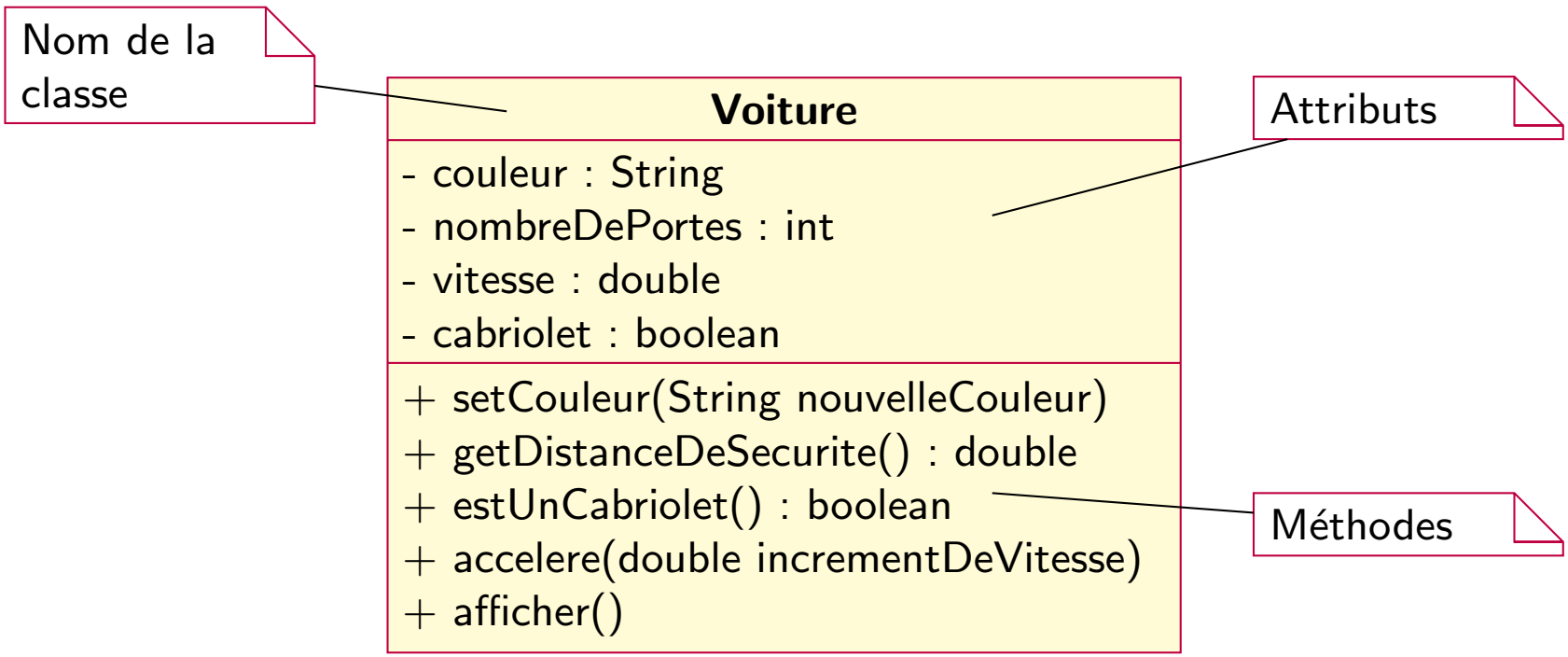


- Une classe est constituée de :
  - Données appelées *attributs*
  - Fonctions/procédures appelées *méthodes*

- Les attributs et les méthodes sont les *membres* de la classe



Qu'est-ce qu'une classe ?

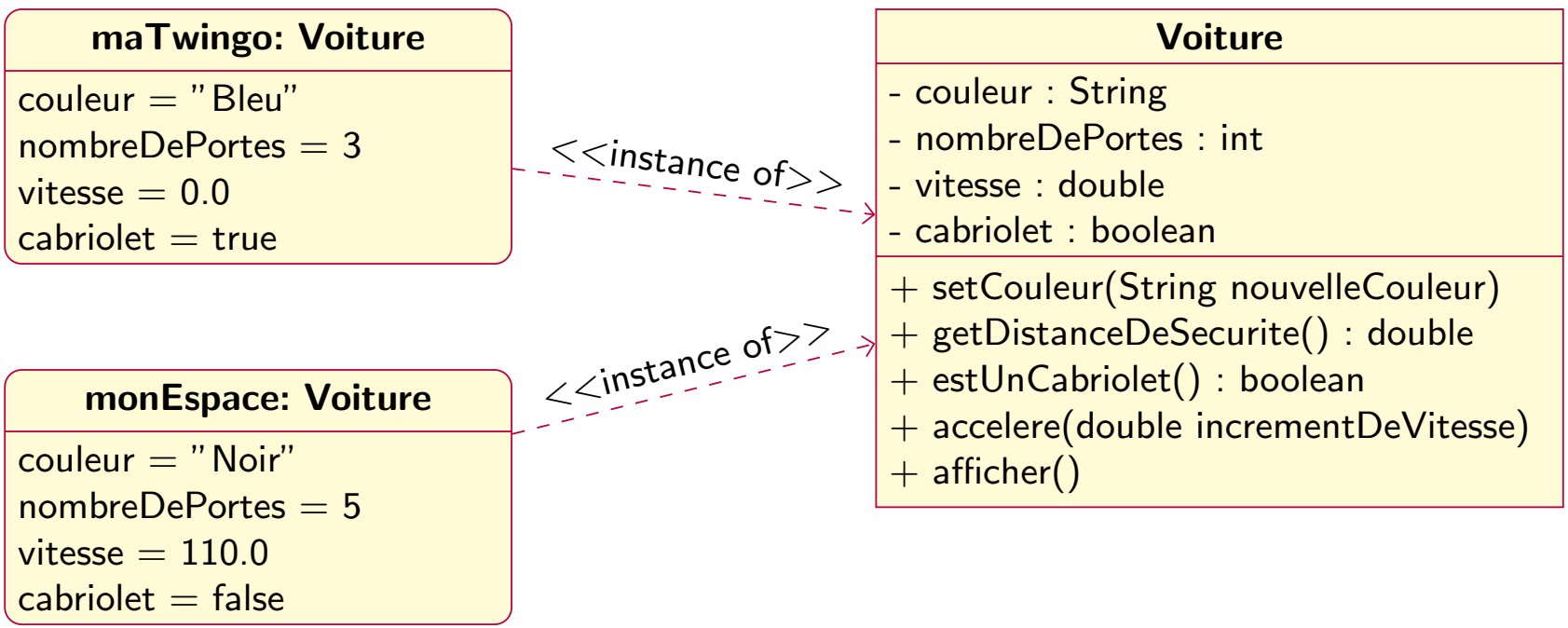


Représentation UML d'une classe

Qu'est-ce que l'instance d'une classe ?

Définition

Un objet est une réalisation concrète d'une classe. On dit qu'un objet est une *instance* d'une classe. La valeur de ses attributs définit son *état*.



Comment définir une classe en Java ?

Définition d'une classe

- Un fichier par classe qui porte obligatoirement le nom de la classe et l'extension `.java`
  Exemple : `Voiture.java`

Définition d'un classe

```
public class NomDeClasse {
    // attributs
    private Classe1 attribut1;
    private Classe2 attribut2;
    ...
    // constructeurs
    public NomDeClasse(arg1,arg2,...) {
        ...
    }
    // méthodes
    public Classe3 méthode1(arg,...) {
        ...
    }
    public Classe4 méthode1(arg,...) {
        ...
    }
    ...
}
```

Définition des attributs d'une classe

- Les attributs d'une classe peuvent être :
  - Des types primitifs : `int`, `double`, `boolean`, etc.
  - Des classes : `String`, `Immatriculation`, `Adresse`, etc.

- Les attributs d'une classe sont *privés* sauf exception

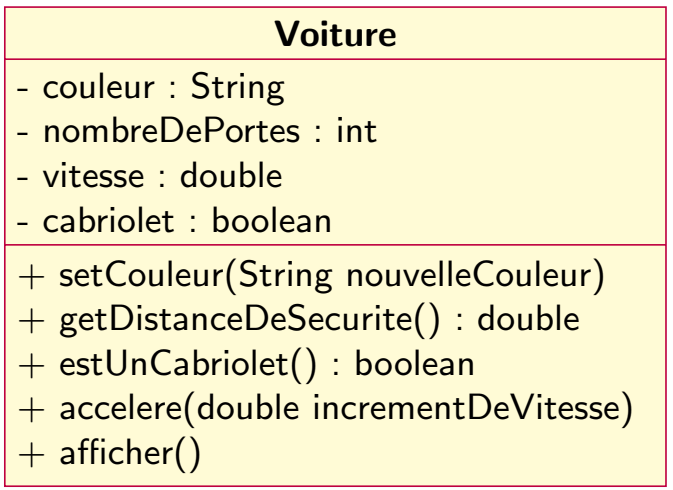
Attributs de la classe Voiture

```
public class Voiture {

    private String couleur;
    private int nombreDePortes;
    private double vitesse;
    private boolean cabriolet;

    ...
}
```

Définition des méthodes d'une classe

- Les méthodes définissent le *comportement* d'un objet

- Les méthodes peuvent être de différentes sortes :
  - Les *constructeurs* initialisent l'état d'un objet
  - Les *accesseurs* renseignent sur l'état d'un objet mais ne l'altère pas
  - Les *modificateurs* permettent d'altérer l'état d'un objet
  - Les *itérateurs* permettent d'accéder à toutes les parties d'un objet dans un ordre bien défini



Définition des accesseurs d'une classe

- Un accesseur (sélecteur, getter) renseigne sur l'état d'un objet mais ne l'altère pas
- Un accesseur est une méthode sans argument
- Un accesseur renvoie un seul résultat à l'aide du mot-clé `return`

Accesseurs de la classe Voiture

```
public class Voiture {
    ...
    public String getCouleur() {
        return this.couleur;
    }

    public double getDistanceDeSecurite() {
        return this.vitesse * 5.0 / 9.0 ;
    }

    public boolean estUnCabriolet() {
        return this.cabriolet;
    }
    ...
}
```

Définition des modificateurs d'une classe

- Un modificateur (mutateur, setter) permettent d'altérer l'état d'un objet
- Un modificateur est une méthode avec un argument ou plus
- Un modificateur ne renvoie pas de résultat

Modificateurs de la classe Voiture

```
public class Voiture {
    ...
    public void setCouleur(String nouvelleCouleur) {
        this.couleur = nouvelleCouleur;
    }

    public void accelere(double incrementDeVitesse) {
        this.vitesse = this.vitesse + incrementDeVitesse;
    }
    ...
}
```

Définition des constructeurs d'une classe

- Un constructeur est une méthode spéciale appelée automatiquement lors de la création d'une instance de la classe (`new`)
- Un constructeur permet d'initialiser les attributs de l'objet nouvellement créé
- On peut définir zéro, un ou plusieurs constructeurs (ils se différencient par le nombre et le type des arguments)
- En l'absence de constructeurs, l'instance est initialisée avec les valeurs par défaut des attributs

Constructeur complet de la classe Voiture

```
public class Voiture {
    ...
    public Voiture(String uneCouleur, int unNombreDePorte, double uneVitesse,
          boolean estUnCabriolet) {
        this.couleur = uneCouleur;
        this.nombreDePorte = unNombreDePorte;
        this.vitesse = uneVitesse;
        this.cabriolet = estUnCabriolet;
    }
    ...
}
```

Définition des constructeurs d'une classe

Constructeur par défaut de la classe Voiture

```
public class Voiture {
    ...
    public Voiture() {
        this.couleur = "Blanc";
        this.nombreDePorte = 5;
        this.vitesse = 0.0;
        this.cabriolet = false;
    }
    ...
}
```

Exemple d'utilisation des constructeurs

```
Voiture maTwingo = new Voiture("Bleu", 3, 0.0, true);
maTwingo.afficher();

Voiture uneVoiture = new Voiture();
uneVoiture.afficher();
```

Définition des constructeurs d'une classe

Constructeur par défaut en utilisant le constructeur complet

```
public class Voiture {
    ...
    public Voiture() {
        this("Blanc",5,0.0,false);
            // possible uniquement si le constructeur complet existe
    }
    ...
}
```

Exemple d'utilisation des constructeurs

```
Voiture maTwingo = new Voiture("Bleu", 3, 0.0, false);
maTwingo.afficher();

Voiture uneVoiture = new Voiture();
uneVoiture.afficher();
```

Définition d'autres méthodes

- De manière générale, on peut définir des méthodes variées avec un ou pas de résultat, zéro ou plusieurs arguments

Méthode afficher

```
public class Voiture {
    ...
    public void afficher() {
        System.out.println("Etat de la voiture");
        System.out.println(" Couleur = " + this.couleur);
        System.out.println(" Nombre de portes = " + this.nombreDePortes);
        System.out.println(" Vitesse = " + this.vitesse);
        System.out.println(" Cabriolet = " + this.cabriolet);
    }
    ...
}
```

Définition d'autres méthodes

- De manière générale, on peut définir des méthodes variées avec un ou pas de résultat, zéro ou plusieurs arguments

Méthode avec une autre voiture

```
public class Voiture {
    ...
    public boolean estPlusRapideQue(Voiture uneVoiture) {
        boolean resultat;
        if (this.vitesse > uneVoiture.vitesse) {
            resultat = true;
        } else {
            resultat = false;
        }
        return resultat;
    }
    ...
}
```

Comment encapsuler les données ?

Comment encapsuler les données ?

- Les membres (attributs et méthodes) d'une classe peuvent être :
  - *Publiques* (+) : dans ce cas, les membres sont visibles et utilisables par les utilisateurs externes à l'objet
  - *Privés* (−) : dans ce cas les membres ne sont pas visibles et utilisables par les utilisateurs externes à l'objet, seul l'accès interne lors de la définition de la classe est possible

Définition

L'encapsulation des données consiste à rendre inaccessibles (privés) tous les attributs d'une classe

Impossible !

```
Voiture maTwingo;
maTwingo = new Voiture(...);
maTwingo.couleur = "Bleu";
maTwingo.afficher() ;
```

Possible !

```
Voiture maTwingo;
maTwingo = new Voiture(...);
maTwingo.setCouleur("Bleu");
maTwingo.afficher() ;
```

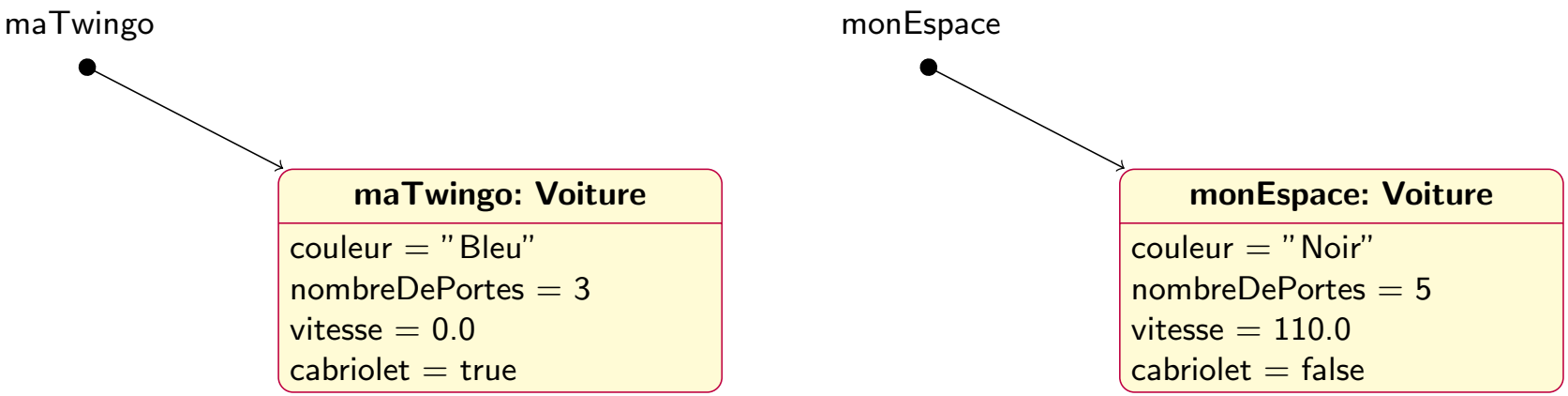
Qu'est ce qu'une référence ?

Qu'est ce qu'une référence ?

- En java, les objets sont manipulés par des références (pointeurs)
- Après déclaration et avant construction la valeur d'une variable est `null`
- L'utilisation d'une variable non construite provoque le lancement d'une exception (erreur) : `NullPointerException`

Exemple

```
Voiture maTwingo;
maTwingo = new Voiture("Bleu", 3, 0.0, true);

Voiture monEspace = new Voiture("Noir", 5, 110, false) ;
```



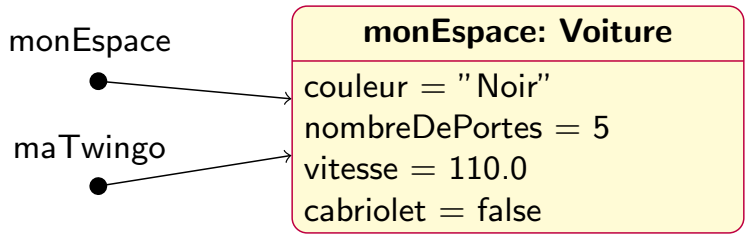
Conséquence pour l'affectation

L'affectation d'un objet à un autre par `a=b;` entraine que :

1. `a` est identique à `b` (même référence)
2. Toute modification de `a` modifie également `b` et inversement

Exemple

```
maTwingo = monEspace;
maTwingo.accelere(20);
```



Solution

Il faut utiliser la méthode `clone()` défini par défaut pour tous les objets afin de dupliquer l'objet en mémoire (nouvelle instance identique à la première) :

```
maTwingo = monEspace.clone();
```

Remarque

Ceci n'est valable que pour les objets ! Pour les variables de types primitifs (`int`, `double`, `boolean`, `char`, etc.), l'affectation `a=b;` recopie bien la valeur de `b` dans `a`

Conséquence pour la comparaison

- Le test de comparaison a==b ne renvoie vrai que si les 2 objets ont la même référence

Exemple 1

```
Voiture maTwingo,monEspace;
maTwingo = new Voiture("Vert", 3, 0.0,
     false);
monEspace = new Voiture("Vert", 3, 0.0,
     false);

if (maTwingo == monEspace) ...
    // FAUX !!
```

Exemple 2

```
Voiture maTwingo,monEspace;
maTwingo = new Voiture("Vert", 3, 0.0,
     false);
monEspace = maTwingo;


if (maTwingo == monEspace) ...
    // VRAI !!
```

Remarque

Ceci n'est valable que pour les objets ! Pour les variables de types primitifs (`int`, `double`, `boolean`, `char`, etc.), le test de comparaison a==b compare bien les valeurs de a et de b

Conséquence pour le passage des arguments

- Pour les types primitifs (`int`, `double`, etc.)
  - La méthode travaille avec une copie de la valeur
  - Les modifications par la méthode n'entraine pas de modification de l'original
- Pour les objets
  - La méthode travaille avec une copie de la référence mais pas des attributs
  - Toute modification de la valeur d'un objet entraine une modification de l'original

Exemple

```
public class Voiture {
    ...

    public void methode1(Voiture uneVoiture) {
        uneVoiture.setCouleur("Bleu");
    }

    public void methode1(int unEntier) {
        unEntier = 5;
    }
    ...
}
```

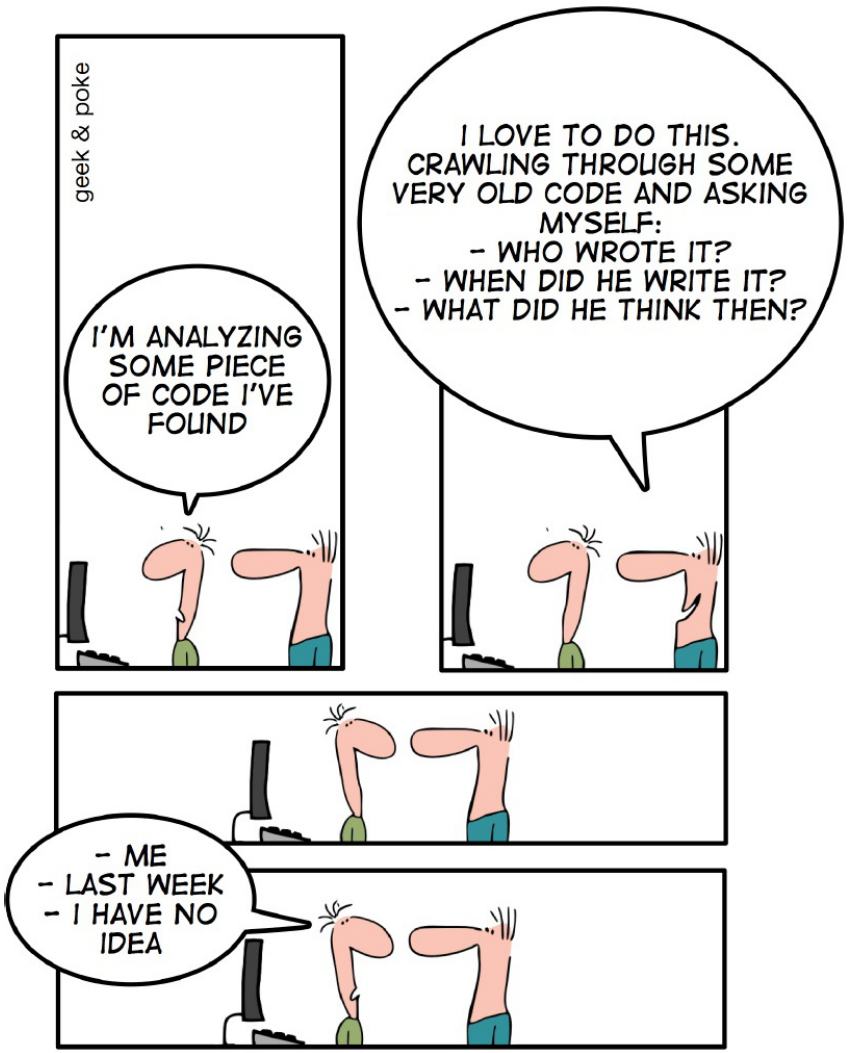
Bonnes pratiques

Bonnes pratiques

- Faire de petites classes
- Faire des méthodes courtes (quelques lignes seulement)
- Le nombre d'arguments d'une méthode ne doit pas dépasser 2
- Si il y a du code dupliqué dans une classe, créer une méthode

Règle n°1

Un code doit être *lisible* !



Comment nommer une classe ?

Règle n°2

L'identificateur d'une classe commence par une majuscule et utilise le *CamelCase*

Règle n°3

L'identificateur d'une classe doit être compréhensible et prononçable

Règle n°4

L'identificateur d'une classe doit donner du sens et circonscrire un domaine

Exemple

```
public class Complexe {

    public double partieReelle;
    public double partieImaginaire;

}
```

Comment nommer un attribut ou une méthode ?

Règle n°5

L'identificateur d'un attribut ou d'une méthode commence par une minuscule et utilise le *camelCase*

Remarques

Quelques habitudes sont devenues conventionnelles :

- les accesseurs commencent par *get*
- les modificateurs commencent par *set*
- les accesseurs booléens commencent par *is* (ou *est*)

Exemple

```
public class Complexe {

    public double partieReelle;
    public double partieImaginaire;

    public void setPartieReelle(double partieReelle);
    public double getModule();
    public boolean estUnImaginairePur();

}
```

Comment nommer une constante ?

Règle n°6

L'identificateur d'une constante est en majuscule, les mots séparés par des tirets bas

Règle n°7

Une constante est déclarée comme un attribut *statique* d'une classe

Exemple

```
public class Math {

    public static double GOLDEN_RATIO = 1.61803398875;

    public static double PI = 3.14159265358979;

}
```

Comment documenter une classe ?

Règle n°8

Utiliser *Javadoc*

Règle n°9

Ne documenter que ce qui nécessite des informations supplémentaires

Exemple

```
/**
 * Classe de simulation d'une voiture
 *
 * @author Mario
 * @version 2.6
 */
public class Voiture {

    /**
     * Calcule la distance de sécurité en mètres
     * selon la règle des 5/9
     */
    public double getDistanceDeSecurite() {
        return vitesse * 5.0 / 9.0;
    }

    public void setCouleur(String nouvelleCouleur) {
        this.couleur=nouvelleCouleur;
    }
}
```

How do I write Clean Code?