# Introduction à la programmation

## M319 — Programmation structurée

Jérôme Frossard

EPAI

8 mars 2026

# Plan

# Programme

En informatique, un **programme** est une **description formelle** d'un système de traitement de données réalisée à l'aide d'un **langage de programmation**.

Un **traitement de données** transforme des données (les **données d'entrée**) pour produire de nouvelles données (les **données de sortie**) et éventuellement un **effet** (affichage, stockage, ou transmission de données, commande d'un actionneur, etc.).

La description d'un système de traitement de données comprend la description des :

- **Algorithmes** (procédures de calcul effectif) qui décrivent les transformations.
- **Données** manipulées par ces algorithmes.
- **Interactions avec le monde extérieur** pour obtenir les données d'entrée et produire un effet observable à l'aide d'**opérations d'entrée/sortie**.

**Attention :** Un programme n'est pas un algorithme. Un algorithme (objet abstrait) peut être mis en œuvre dans un programme (objet concret). De plus, contrairement à un algorithme, un programme peut ne pas se terminer.

# Langage de programmation

Un langage de programmation est une **langue artificielle** qui permet de décrire sous forme de **texte** les structures de données, les algorithmes et les interactions avec le monde qui constituent un **programme** informatique.

Il doit être à la fois :

- Facile à lire et écrire par un·e programmeur·euse (un langage de programmation doit permettre de communiquer des idées).
- Adapté au traitement par une machine (exécution par un interpréteur, traduction par un compilateur, coloration syntaxique par l'éditeur, linter, etc.)

Pour cela, un langage de programmation utilise souvent des **mots-clés** empruntés à l'anglais (`if`, `else`, `while`, `do`, `try`, etc.) et adopte une **syntaxe formelle**, parfois inspirée de notations mathématiques (p. ex. la notation algébrique pour les expressions arithmétiques).

# Compilateur

Un **compilateur** (*compiler*) est un programme qui **traduit** un programme écrit avec un **langage source** en un programme sémantiquement équivalent dans un **langage cible**.

Le langage source est souvent un langage de programmation. Lorsque c'est le cas, le texte du programme est appelé **code source** (*source code*).

Le langage cible peut être :

- Un langage machine (AMD64, AArch64, etc.)
- Un langage intermédiaire (.NET Common Intermediate Language, Java bytecode, etc.)
- Un autre langage de programmation (JavaScript, C, etc.)

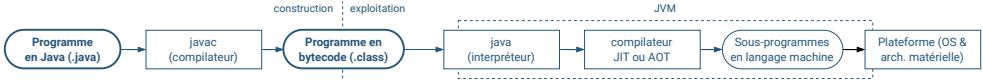
Dans le développement web, JavaScript est le langage supporté par les navigateurs. C'est pourquoi il constitue une cible pour les compilateurs de plusieurs langages (TypeScript, Dart, PureScript, etc.).

**Remarque :** Un langage intermédiaire ressemble souvent à un langage machine : le code du programme n'est pas un texte, mais une longue séquence de nombres. C'est le cas des langages intermédiaires de .NET et Java.

# Interpréteur

Un **interpréteur** (*interpreter*) est un programme qui **exécute un programme** écrit dans un langage de programmation ou un langage intermédiaire.

Compilateur et interpréteur ne s'excluent pas mutuellement. Bien souvent, un interpréteur utilise un compilateur pour traduire le programme en langage machine juste avant de l'exécuter. Selon les cas, il peut s'agir d'un compilateur *just-in-time* (*JIT compiler*), ou d'un compilateur *ahead of time* (*AOT compiler*).




Par exemple, en Java :

- Le code source est compilé en bytecode avec le compilateur `javac`.
- Le programme en bytecode est exécuté avec l'interpréteur `java`
- La JVM utilise typiquement un compilateur JIT ou AOT pour traduire le bytecode dans le langage machine de la plateforme (au fur et à mesure ou à l'avance)

# Plan

# Variables et constantes (I/II)

Les données qui doivent être décrites dans un programme sont en général des **données du monde réel** :

- La valeur de certaines données ne change jamais (p. ex. une constante mathématique). Dans un programme, ces données sont représentées par des **constantes**.
- Celle des autres peut varier d'une exécution à l'autre. Dans un programme, ces données sont représentées par des **variables**.

La valeur d'une constante est en principe écrite directement dans le code (**valeur littérale**), celle d'une variable dépend souvent, directement ou indirectement, d'une interaction avec le monde réel.

Une donnée n'est pas juste une valeur. Elle **représente quelque chose** (la description d'un cours, la température de la classe, l'état des boutons de la souris, etc.) et elle **a une certaine nature**, un type (une chaîne de caractères, un nombre décimal, un nombre entier, etc.).

# Variables et constantes (II/II)

Dans un langage de programmation, une variable ou une constante a **trois attributs** :

- Un **identificateur** (un nom) : indique ce qu'est la donnée représentée
- Une **valeur** : la valeur de la donnée
- Un **type** de données : indique la nature de la donnée

En **programmation impérative**, la valeur d'une variable peut être **modifiée (mutée) en cours d'exécution**. C'est le cas, par exemple, du compteur d'une boucle `for`, d'un accumulateur, ou encore de certaines variables non locales.

Toutefois, il est important de noter que :

- Ce n'est pas ce qui définit une variable de manière générale (pas de mutation en math, pas de mutation en programmation fonctionnelle)
- Même si sa valeur change, une variable devrait **toujours représenter la même donnée**. Changer la signification d'une variable nuit à la lisibilité du programme.

# Type de données

Un **type de données** permet de décrire la **nature d'une donnée**.

Par exemple, si une variable représente une température en degrés Celsius (°C), sa valeur est probablement un nombre décimal. Dans un langage de programmation, on peut modéliser cette valeur avec le type `float` (*IEEE 754 single precision*) qui permet de représenter des nombres décimaux avec une précision d'environ 7 chiffres significatifs.

Toutefois, le type `float` supporte la division qui n'est pas une opération valide sur une échelle d'intervalles, et ne permet pas de connaître l'unité de la valeur (°C, °F, K, etc.). On pourrait donc imaginer un type `Temperature` qui permet d'indiquer l'unité, limite les opérations arithmétiques, et définit, par exemple, une opération de conversion.

De manière générale, un type de données est un **attribut d'une donnée** qui définit un :

- **Ensemble de valeurs** que peut prendre cette donnée ou la **structure** de cette donnée.
- **Ensemble d'opérations** applicables à ces valeurs ou aux instances de cette structure.

# Types simples et types composés

La plupart des langages de programmation ont des **types simples** prédéfinis qui modélisent des nombres entiers (signés ou non), des nombres décimaux, des booléens, des caractères ou encore des chaînes de caractères. Pour ces types, il existe en principe une notation permettant d'écrire une valeur fixe dans le code (**valeur littérale**).

En Java, ces types sont appelés types primitifs : `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` et `char`. La chaîne de caractères (`String`) n'est pas un type primitif, mais il y a une notation pour les valeurs littérales de chaîne.

Les **types composés** comprennent les tableaux et les types définis par l'utilisateur. Ces derniers sont définis au moyen de constructions syntaxiques du langage à partir de types simples ou d'autres types composés.

En Java, un type défini par l'utilisateur est réalisé à l'aide d'une classe ou d'une interface. Les types `Scanner`, `PrintStream`, `InputStream`, `ArrayList`, etc. sont des types définis dans la bibliothèque standard de Java.

# Sous-programme, paramètres et arguments

Pour **faciliter la lecture** et **éviter la répétition de code**, un programme est généralement subdivisé en plusieurs **sous-programmes**. On en distingue deux sortes :

- Les **fonctions** qui produisent (renvoient) une valeur.
- Les **procédures** qui produisent un effet, mais pas de valeur.

Un sous-programme est une portion de code associée à un **nom**, une **liste de paramètres** (possiblement vide), et le type de la valeur produite si c'est une fonction. Le code du sous-programme est exécuté lorsqu'il est **appelé** par son nom avec une **liste d'arguments**.

Les paramètres représentent les données d'entrée du sous-programme :

- Les **arguments** de l'appel sont des expressions dont la valeur est liée aux paramètres.
- Le plus souvent, les arguments sont **liés aux paramètres par position** (le 1er argument est lié au 1er paramètre, le 2e argument au 2e paramètre, etc.).
- Dans le sous-programme, un paramètre est exactement **comme une variable locale**.

# Plan

# Programmation impérative

La **programmation impérative** est un paradigme de programmation (une manière de concevoir la programmation) dans lequel un programme est une **séquence d'instructions** (des ordres exprimés à l'impératif) que la machine doit **exécuter** pour parvenir au résultat.

Chaque **instruction** est une étape vers le résultat et doit donc **modifier l'état du système** : elle doit **produire un effet**. Une instruction qui n'a pas d'effet ne sert à rien. On parle de **séquence d'instructions** pour indiquer que les instructions doivent être **exécutées dans l'ordre** : l'effet d'une instruction peut dépendre de l'effet de la précédente.

Les instructions de base sont :

- Affectation (initialise ou modifie la valeur d'une variable)
- Branchement conditionnel (*if-goto*) et non-conditionnel (*goto*)
- Appel de sous-programme (*call*)

**Remarque :** L'appel de sous-programme est un cas particulier de branchement non conditionnel (voir Appel de sous-programme).

# Programmation structurée (I/II)

En programmation impérative, un branchement **contrôle le flux** d'instructions en modifiant le registre PC (*program counter*) qui contient l'adresse de la prochaine instruction. Si le branchement est conditionnel, le PC n'est modifié que si la condition est vérifiée.

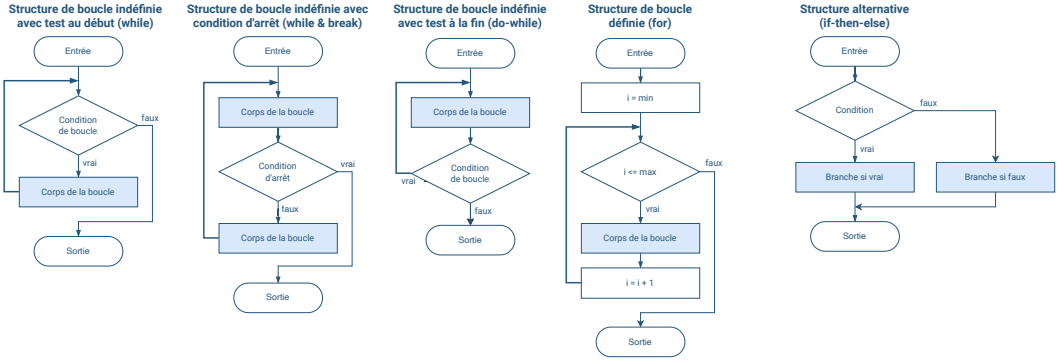
La programmation structurée est un style de programmation dans le paradigme impératif, qui propose de n'utiliser les branchements que pour réaliser des **structures de contrôle** avec **une seule entrée** et **une seule sortie**. Le but est de rendre le code le plus facile à lire grâce à des **motifs facilement identifiables**.

Les principales structures de contrôle sont :

- Structure de boucle définie (boucle à compteur).
- Structure de boucle indéfinie (boucle conditionnelle).
- Structure alternative.

**Remarque :** Une forme primitive de boucles à compteur existe déjà dans la première version de FORTRAN (1954) et dans le Dartmouth BASIC (1964). La programmation structurée apparaît dès la fin des années 1950, mais reste cantonnée au milieu académique jusque dans les années 1970 avec la diffusion de Pascal et de C.

# Programmation structurée (II/II)




Avec un langage de programmation qui supporte la programmation structurée, le respect des **contraintes** ne repose plus sur la discipline de la programmeuse ou du programmeur, mais est **imposé par la syntaxe**. Chaque structure de contrôle est une construction ad-hoc : `while`, `do-while`, `while` avec `break`, `for`, `if-then-else`, etc.

# Plan

# Identificateur

Dans un langage de programmation, un **identificateur** est un nom (une étiquette, un symbole) qui **désigne une entité du programme** : variable, sous-programme, type, etc.

Typiquement, un identificateur doit commencer par une lettre ou un tiret du bas (*underscore*), et être composé de lettres, de chiffres, et de tirets du bas, sans espaces.

Pour nommer les entités du programme, on utilise souvent les conventions suivantes :

- Un groupe nominal pour une variable ou un type (p. ex. numberOfItems, Scanner)
- Un groupe verbal dans le cas d'un sous-programme (p. ex. findItem)

Pour séparer les mots, on utilise les conventions de nommage suivantes :

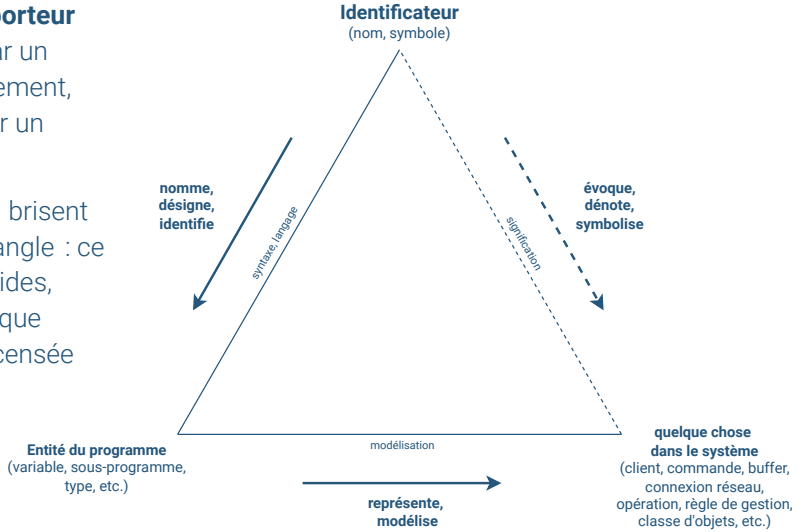
- UpperCamelCase et lowerCamelCase
- UPPER_SNAKE_CASE et lower_snake_case

Chaque langage (ou famille de langages) utilise ces conventions à sa manière. Le respect des conventions du langage facilite la lecture en réduisant la surprise.

# Sens d'un identificateur

Un identificateur doit être **porteur de sens** : explicitement (par un nom descriptif) ou implicitement, par convention (p. ex. i pour un compteur de boucle).

Des noms comme f1 ou v1 brisent l'axe « signification » du triangle : ce sont des identificateurs valides, mais ils n'évoquent pas ce que l'entité du programme est censée représenter.



# Déclaration et définition des identificateurs

La **déclaration** d'un identificateur est une construction syntaxique qui nous permet d'indiquer au compilateur que l'on a l'**intention d'utiliser** cet identificateur.

La **définition** d'un identificateur est une construction syntaxique qui permet de **déclarer** cet identificateur et de lui **associer ce à quoi correspond l'entité** qu'il désigne : la valeur dans le cas d'une variable, la liste de paramètres et le corps dans le cas d'un sous-programme, la structure de données dans le cas d'un type, etc.

Si le langage exige la déclaration des identificateurs (c'est le cas de Java), un compilateur peut émettre un avertissement ou une erreur lorsque :

- Un identificateur est déclaré, mais pas utilisé (identificateur inutile ou erreur de programmation).
- Un identificateur est utilisé, mais pas déclaré (oubli ou identificateur mal orthographié).

Avec un langage statiquement typé, le type d'un identificateur doit être explicitement spécifié s'il ne peut pas être inféré.

# Portée lexicale d'un identificateur

En simplifiant un peu, on peut définir la **portée lexicale (*lexical scope*)** d'un identificateur comme la **portion du code** dans laquelle cet identificateur peut être référencé (utilisé) :

- Un sous-programme (portée locale)
- Un module, une classe, ou un espace de noms.
- L'ensemble du programme (portée globale)

Dans certains langages, la portée d'un identificateur peut également être un bloc dans un sous-programme (portée de bloc).

En Java, dans une méthode, la portée d'un identificateur est le bloc (délimité par des accolades) dans lequel il est déclaré. La portée des identificateurs des membres d'une classe (variables et méthodes) est le corps de la classe, mais leur portée apparente dépend de leur **visibilité** (public, private, etc.) et de la portée de l'identificateur de la classe.

**Attention :** Pour une variable, portée lexicale (programmation) et durée de vie (exécution) sont deux notions distinctes qui ne doivent pas être confondues.

# Expression (produire une valeur)

Dans un langage de programmation, une **expression** est une construction syntaxique qui peut être **évaluée** pour **produire une valeur**.

Une expression peut être :

- Une constante littérale (une valeur écrite dans le texte du programme).
- Le nom d'une variable.
- Une opération dont les opérandes sont des expressions.
- Un appel (ou application) de fonction dont les arguments (paramètres effectifs) sont des expressions.

L'utilisation du terme expression dans cette définition en fait une **définition récursive**. Elle peut ainsi s'appliquer à des expressions arbitrairement complexes.

**Remarque :** Idéalement, une expression ne devrait pas avoir d'effet de bord ni dépendre d'un état externe mutable (variable non locale, E/S, etc.). C'est toutefois courant et parfois inévitable en programmation impérative, même si on considère que c'est une bonne pratique de l'éviter autant que possible.

# Instruction (produire un effet)

Dans un langage de programmation impératif, une **instruction** est une construction syntaxique qui peut être **exécutée** pour **produire un effet**.

Une instruction peut être, notamment :

- Une affectation dont la partie droite est une expression.
- Un appel de procédure dont les arguments (paramètres effectifs) sont des expressions.
- Une structure de choix dont la condition est une expression booléenne et chaque branche est une instruction.
- Une structure de boucle dont la condition est une expression booléenne et le corps est une instruction.
- Une **séquence d'instructions**.

Cette définition est également récursive. Elle s'applique donc, elle aussi, à des instructions arbitrairement complexes.

# Plan

# Pile d'appels (call stack)

Lors de l'exécution d'un programme, chaque appel de sous-programme doit créer un nouveau **contexte d'exécution** afin de préserver celui du sous-programme appelant.

Ce contexte d'exécution comprend notamment :

- Les arguments (valeur des paramètres) passés lors de l'appel.
- L'adresse de retour (où reprendre l'exécution dans l'appelant).
- Les variables locales du sous-programme.
- D'autres informations nécessaires à l'exécution (registres sauvegardés, etc.).

Pour cela, on utilise généralement une **pile d'appels** — une structure de données de type LIFO (*Last In, First Out*), à l'image d'une pile d'assiettes. Un **nouveau contexte** est **empilé au moment de l'appel** d'un sous-programme, puis dépilé lorsqu'il se termine.

Quand les appels sont imbriqués, le dernier sous-programme appelé (*Last In*) est bien le premier à se terminer (*First Out*).

# Durée de vie d'une variable

La **durée de vie** d'une variable est la période durant laquelle une variable **existe dans la mémoire** de la machine qui exécute le programme.

En général, les variables locales font partie du contexte d'exécution du sous-programme. Leur durée de vie est donc, au plus, celle de ce contexte.

Par contraste, et en simplifiant un peu, une variable avec le modificateur `static` existe dans un segment de données (*data segment*), une portion de la mémoire du processus dédiée aux variables allouées de manière statique. Celles-ci existent donc en principe durant le temps d'exécution du programme.

La **portée** de l'identificateur **ne détermine pas la durée de vie** de la variable. Celle-ci peut exister dans la mémoire, même si son identificateur n'est pas accessible dans la portion de code en cours d'exécution. C'est notamment le cas des variables statiques.

La durée de vie exacte d'une variable dépend du modèle d'exécution décrit dans la spécification du langage et peut donc varier d'un langage à l'autre.

# Appel de sous-programme (transfert de contrôle)

Lors d'un appel, le sous-programme appelant **transfère le contrôle** au sous-programme appelé : il dit au CPU d'interrompre l'exécution de la séquence d'instructions de l'appelant pour exécuter celle de l'appelé, puis continuer là où il s'était interrompu.

Pour cela, il faut :

- Créer un nouveau contexte (*stack frame*) dans la pile d'appels (*call stack*) et y placer les arguments (l'ordre dépend de la convention du langage).
- Enregistrer l'**adresse de retour** (l'adresse de l'instruction qui suit l'appel).
- Modifier le registre PC (*program counter*) pour pointer vers la première instruction du sous-programme appelé : **branchement non conditionnel**.

L'instruction `call` combine généralement les deux dernières opérations.

Lorsque l'exécution du sous-programme se termine, la valeur de retour est, le cas échéant, passée via la pile ou un registre selon la plateforme, le stack frame est détruit, et le registre PC est modifié pour pointer vers l'adresse de retour.

# Plan

# Point d'entrée du programme

Le **point d'entrée** d'un programme est le sous-programme qui contient la **première instruction du programme**. Il s'agit souvent d'une fonction ou d'une procédure **main** (Java, C#, etc.), ou du fichier passé à l'interpréteur (JavaScript, Python, etc.).

La particularité du point d'entrée est d'être appelé par une entité extérieure au programme (OS, JVM, interpréteur, etc.). Son exécution, en revanche, n'a rien de particulier :

- L'entité appelante passe le contrôle au point d'entrée (création du stack frame, passage des arguments, etc.)
- Lorsque l'exécution se termine, il rend le contrôle à l'appelant (éventuellement avec un code de retour), typiquement avec une instruction `return`.

Dans beaucoup de langages, une instruction `exit` permet de rendre immédiatement le contrôle à l'entité appelante.