Perl 6

1. Introduction

1-1. Perl 6, c'est quoi?

Perl 6 est un langage de haut niveau, générique et dynamique. Il supporte plusieurs paradigmes dont : la programmation procédurale, la programmation orientée objet et la programmation fonctionnelle.

La devise de Perl 6 :

- TMTOWTDI (prononcé Tim Toady) : *There is more than one way to do it*: c'est-à-dire « Il y a plus d'une façon de le faire ».
- *Easy things should stay easy, hard things should get easier, and impossible things should get hard*: « Les choses faciles doivent rester faciles, les choses difficiles devraient devenir plus faciles, et les choses impossibles devraient devenir difficiles ».

Un programme ou script Perl 6 est un fichier texte qui sera compilé et exécuté par l'exécutable `perl6` et la machine virtuelle associée (par exemple MoarVM ou JVM).

1-2. Jargon

- **Perl 6** est une spécification de langage avec une suite de tests. Les implémentations qui passent la suite de tests sont considérées comme du Perl 6.
- **Rakudo** est un compilateur pour Perl 6.
- **Rakudobrew** est un gestionnaire d'installation pour Rakudo.
- **Panda** est un installeur de modules pour Perl 6.
- **Rakudo Star** est un paquet qui comprend : Rakudo, Panda, une collection de modules, et de la documentation.

1-3. Installer Perl 6

Linux

1. Installez Rakudo : tapez la commande suivante dans le terminal `rakudobrew build moar`.
2. Installez Panda : tapez la commande suivante dans le terminal `rakudobrew build panda`.

OS X

Quatre solutions possibles :

- Suivez les mêmes étapes que celles indiquées pour l'installation sur Linux, ou
- Installez avec homebrew : `brew install rakudo-star`, ou
- Installez avec MacPorts : `sudo port install rakudo`, ou

Windows

1. Après l'installation, ajoutez `C:\rakudo\bin` à votre PATH.

Docker

1. Obtenez l'image officielle `docker pull rakudo-star`.
2. Ensuite exécutez `docker run -it rakudo-star`.

1-4. Exécuter du code Perl6

On peut exécuter du code Perl 6 en utilisant le terminal Perl 6 interactif *REPL* (Read-Eval-Print Loop). Pour ce faire, ouvrez un terminal, tapez `perl6` dans la fenêtre de terminal et ensuite la touche [Entrée].
Une invite de commande `>` apparaîtra.
Ensuite, tapez une ligne de code puis la touche [Entrée]. Le REPL affichera la valeur de la ligne interprétée. Vous pouvez taper une autre ligne, ou `exit` et ensuite [Entrée] pour sortir du REPL.

L'autre façon consiste à écrire votre code dans un fichier, le sauvegarder puis l'exécuter. Il est conseillé, pour plus de clarté, que les scripts Perl 6 portent l'extension `.pl6`. Exécutez le fichier en entrant `perl6 nom-du-fichier.pl6` dans la fenêtre de terminal (puis [Entrée]). À l'inverse du REPL, le résultat ne sera pas automatiquement affiché pour chaque ligne : le code doit contenir une instruction comme `say` ou `print` pour afficher une sortie.

Le REPL est la plupart du temps utilisé pour essayer un morceau de code, le plus souvent une seule ligne. Pour des programmes de plus d'une seule ligne, la méthode fichier/exécution est recommandée.

Les lignes de code unilignes peuvent aussi être entrées de façon non interactive sur la ligne de commande en tapant `perl6 -e 'mon code ici'` et ensuite [Entrée].

Rakudo Star fournit un éditeur ligne par ligne, qui augmente les fonctionnalités du REPL. Comme : le rappel des commandes par les flèches « haut/bas », l'édition avec les flèches « gauche/droite » et la complétion avec la touche [TAB].

Si vous avez seulement installé Rakudo au lieu de Rakudo Star, vous n'aurez probablement pas les fonctions d'édition ligne par ligne. Lancez la commande suivante sur votre terminal pour y avoir accès :

- `panda install Linenoise` : fonctionne sur Windows, Linux et OS X
- `panda install Readline` : si vous êtes sur Linux et préférez la bibliothèque Readline

1-5. Éditeurs

Comme la plupart du temps, nous allons écrire et stocker nos programmes en Perl 6 dans des fichiers, nous devrions avoir un éditeur de texte décent qui reconnaît la syntaxe de Perl 6.

Je recommande personnellement **Atom**. C'est un éditeur de texte moderne livré avec une coloration syntaxique pour Perl 6. **Perl6-fe** est une coloration syntaxique alternative pour Perl 6 sur Atom, basée sur l'originale, mais comprenant de nombreux correctifs et ajouts.

D'autres personnes de la communauté utilisent aussi **Vim**, **Emacs** ou **Padre**.

Les versions récentes de Vim sont livrées avec la coloration syntaxique pour Perl 6. Emacs et Padre nécessiteront l'installation de paquets supplémentaires.

N'importe quel éditeur de texte peut être utilisé pour écrire ou lire un programme Perl ou Perl 6.

1-6. Bonjour Monde !

Nous allons commencer avec le rituel `hello world`.

```
say 'Bonjour Monde';
```

qui peut aussi s'écrire:

```
'Bonjour Monde'.say;
```

1-7. Aperçu de la syntaxe

Perl 6 a une **forme libre** : vous êtes libre (la plupart du temps) d'utiliser n'importe quelle quantité d'espaces.

Les **instructions** sont typiquement une ligne logique de code, elles doivent se terminer par un point-virgule : `say "Hello" if True;`

Les **expressions** sont un type spécial d'instructions qui retournent une valeur : `1+2` retourne `3`

Les expressions sont faites de **termes** et d'**opérateurs**.

Les **termes** sont des :

- **variables** : une valeur qui peut être manipulée ou changée ;
- **valeurs littérales** : une valeur constante comme un nombre ou une chaîne.

Les **opérateurs** sont classés en types :

Type	Explication	Exemple
Préfixé	Avant le terme	++1
Infixé	Entre deux termes	1+2
Suffixé	Après le terme	1++
Circonfixé	Autour du terme	(1)
Postcirconfixé	Après un terme, autour d'un autre	Array[1]

1-7-1. Identificateurs

Les identificateurs sont le nom donné aux termes lors de leur définition.

Règles :

- ils doivent commencer par un caractère alphabétique ou un tiret bas (*underscore*) ;
- ils peuvent contenir des chiffres (à l'exception du premier caractère) ;

- ils peuvent contenir des tirets ou des apostrophes (sauf le premier et le dernier caractère), mais avec un caractère alphabétique à droite de chaque tiret apostrophe.

Valide	Non valide
var1	1var
var-one	var-1
var'one	var'1
var1_	var1'
_var	-var

Conventions de nommage :

- Camel: `variableNo1`
- Kebab: `variable-no1`
- Snake: `variable_no1`

Vous êtes libre de nommer vos identificateurs comme vous le souhaitez, mais, pour des raisons de cohérence et de lisibilité, il est recommandé de choisir une convention de nommage et de s'y tenir.

L'utilisation de noms signifiants facilitera votre vie et celle des autres. `var1 = var2 * var3` est syntaxiquement correct, mais son but n'est pas évident. `salaire-mensuel = salaire-journalier * jours-travaillés` serait une meilleure façon de nommer vos variables.

1-7-2. Commentaires

Un commentaire est du texte ignoré par le compilateur.

Il y a trois types de commentaires :

- ligne unique :

```
#Ceci est une seule ligne de commentaire
```

- intégré :

```
say #`(Ceci est un commentaire intégré)
"Bonjour Monde."
```

  Le caractère « (» après `#`` définit le début du commentaire, le commentaire se termine avec une parenthèse fermante. On peut de même utiliser des crochets ou accolades, ou même des combinaisons de ces caractères.

- multiligne:

```
=begin comment
Ceci est un commentaire sur plusieurs lignes.
Commentaire 1
Commentaire 2
=end comment
```

1-7-3. Guillemets

Les chaînes doivent être délimitées par des guillemets droits, doubles ou simples (apostrophes).

Utilisez toujours des guillemets droits doubles :

- si votre chaîne contient une apostrophe ;
- si votre chaîne contient une variable qui doit être interpolée.

```
say 'Bonjour Monde';              # Bonjour Monde
say "Bonjour Monde";              # Bonjour Monde
say "Quelqu'un m'a dit";          # Quelqu'un m'a dit
my $nom = 'François Pinon';
say 'Salut $nom';                 # Salut $nom
say "Salut $nom";                 # Salut François Pinon
```

Il est cependant possible de « protéger » un guillemet dans une chaîne entre guillemets (ou une apostrophe dans une chaîne entre apostrophes) à l'aide du caractère d'échappement `\` :

```
say 'Quelqu\'un m\'a dit';        # Quelqu'un m'a dit
```

2. Opérateurs

Opérateur	Type	Description	Exemple	Résultat
+	Infixé	Addition	1 + 2	3
-	Infixé	Soustraction	3 - 1	2
*	Infixé	Multiplication	3 * 2	6
**	Infixé	Puissance	3 ** 2	9
/	Infixé	Division	3 / 2	1.5
div	Infixé	Division entière (arrondit vers le bas)	3 div 2	1
%	Infixé	Modulo (reste de la division entière)	7 % 4	3
%%	Infixé	Divisibilité	6 %% 4	False
			6 %% 3	True
gcd	Infixé	Plus grand dénominateur commun	6 gcd 9	3
lcm	Infixé	Plus petit commun multiple	6 lcm 9	18
==	Infixé	Égalité	9 == 7	False
!=	Infixé	Inégalité	9 != 7	True
<	Infixé	Plus petit	9 < 7	False
>	Infixé	Plus grand	9 > 7	True
<=	Infixé	Plus petit ou égal	7 <= 7	True
>=	Infixé	Plus grand ou égal	9 >= 7	True
eq	Infixé	Égalité (chaînes)	"Tintin" eq "Tintin"	True
ne	Infixé	Inégalité (chaînes)	"Tintin" ne "Titine"	True

=	Infixé	Affectation	my $var = 7	Attribue la valeur 7 a la variable $var
~	Infixé	Concaténation	9 ~ 7	97
			"Bonjour " ~ "chez vous"	Bonjour chez vous
x	Infixé	Réplication	13 x 3	131313
			"Salut " x 3	Salut Salut Salut
~~	Infixé	Smart match (reconnaissance intelligente)	2 ~~ 2	True
			2 ~~ Int	True
			"Perl 6" ~~ "Perl 6"	True
			"Perl 6" ~~ Str	True
			"Renaissance" ~~ /naissance/	「naissance」
++	Préfixé	Incrémentation	my $var = 2; ++$var;	Incrémente la variable de 1 et retourne le résultat 3
	Suffixé	Incrémentation	my $var = 2; $var++;	Retourne la variable 2 et puis l'incrémente
--	Préfixé	Décrémentation	my $var = 2; --$var;	Décrémente la variable de 1 et retourne le résultat 1
	Suffixé	Décrémentation	my $var = 2; $var--;	Retourne la variable 2 et puis la décrémente
+	Préfixé	Force l'opérande à une valeur numérique	+"3"	3
			+True	1
			+False	0
-	Préfixé	Force l'opérande à une valeur numérique et retourne la négation	-"3"	-3
			-True	-1
			-False	0
?	Préfixé	Force l'opérande à une valeur booléenne	?0	False
			?9.8	True
			?"Hello"	True
			?""	False
			my $var; ?$var;	False
			my $var = 7; ?$var;	True
!	Préfixé	Force l'opérande à une valeur booléenne et retourne la négation	!4	False
..	Infixé	Construction d'intervalles	0..5	Crée un intervalle de 0 à 5
..^	Infixé	Construction d'intervalles	0..^5	Crée un intervalle de 0 à 4
^..	Infixé	Construction d'intervalles	0^..5	Crée un intervalle de 1 à 5
^..^	Infixé	Construction d'intervalles	0^..^5	Crée un intervalle de 1 à 4

^	Préfixé	Construction d'intervalles	^5	Comme 0..^5 Crée un intervalle de 0 à 4
…	Infixé	Construction de listes paresseuses	0…9999	Retourne les éléments seulement si nécessaire
|	Préfixé	Aplanissement	|(0..5)	(0 1 2 3 4 5)
			|(0^..^5)	(1 2 3 4)

3. Variables

Les variables sont classées en trois catégories : scalaires, tableaux et hachages.

Un **sigil** (signe en latin) est un caractère utilisé comme préfixe pour classer les variables.

- `$` est utilisé pour les scalaires.
- `@` est utilisé pour les tableaux.
- `%` est utilisé pour les tables de hachage.

3-1. Scalaire

Un scalaire contient une valeur ou une référence.

```
#String
my $nom = 'François Pinon';
say $nom;

#Integer
my $age = 20;
say $age;
```

Certaines opérations peuvent être effectuées sur un scalaire, suivant le type de valeur qu'il contient.

Chaîne

```
my $nom = 'François Pinon';
say $nom.uc;
say $nom.chars;
say $nom.flip;
```

ce qui affiche :

```
FRANÇOIS PINON
14
noniP sioçnarF
```

Entier

```
my $age = 17;
say $age.is-prime;
```

Ce qui affiche :

```
True
```

Pour une liste exhaustive des méthodes applicables aux entiers, voir **http://doc.perl6.org/type/Int**.

Nombre rationnel

```
my $age = 2.3;
say $age.numerator;
say $age.denominator;
say $age.nude;
```

Ceci affiche :

```
23
10
(23 10)
```

3-2. Tableaux

Les tableaux sont des listes contenant plusieurs valeurs. Par défaut, les valeurs d'un tableau peuvent être de différents types.

```
my @animaux = 'chameau','lama','hibou';
say @animaux;
```

De nombreuses opérations peuvent être effectuées sur les tableaux comme le montre l'exemple suivant :

Le tilde ~ est utilisé pour la concaténation.

```
my @animaux = 'chameau','vigogne','lama';
say "Le zoo contient " ~ @animaux.elems ~ " animaux";
say "Les animaux sont: " ~ @animaux;
say "Je vais adopter un hibou pour le zoo";
@animaux.push("hibou");
say "Maintenant, mon zoo contient: " ~ @animaux;
say "Le premier animal que nous avons adopté est le " ~ @animaux[0];
@animaux.pop;
say "Malheureusement, le hibou est parti, il ne nous reste que: " ~
@animaux;
say "Nous allons fermer le zoo et laisser un animal seulement";
say "Nous allons faire partir: " ~ @animaux.splice(1,2) ~ " et
laisser le " ~ @animaux;
```

Sortie

```
Le zoo contient 3 animaux
Les animaux sont: chameau vigogne lama
Je vais adopter un hibou pour le zoo
Maintenant, mon zoo contient: chameau vigogne lama hibou
Le premier animal que nous avons adopté est le chameau
Malheureusement, le hibou est parti, il ne nous reste que: chameau
vigogne lama
Nous allons fermer le zoo et laisser un animal seulement
Nous allons faire partir: vigogne lama et laisser le chameau
```

Explication

`.elems` retourne le nombre d'éléments contenus dans le tableau.
`.push()` ajoute un élément au tableau.
Nous pouvons accéder à un élément spécifique dans le tableau en spécifiant sa position `@animaux[0]`.
`.pop` supprime le dernier élément du tableau.
`.splice(a,b)` supprime les b éléments à partir de la position a.

3-2-1. Tableaux de taille fixe

Un tableau simple se déclare comme ceci :

```
my @tableau;
```

Le tableau simple à une taille non définie, et peut varier de façon automatique. Ce tableau acceptera un nombre illimité de valeurs sans restriction.

On peut en revanche créer des tableaux de taille fixe.
Ces tableaux ne pourront pas excéder la taille qui leur aura été allouée (en lecture et écriture).

Pour déclarer un tableau de taille fixe, spécifiez son nombre maximal d'éléments entre crochets à la suite de son nom :

```
my @tableau[3];
```

Ce tableau pourra contenir un maximum de trois valeurs, indexées de 0 à 2.

```
my @tableau[3];
@tableau[0] = "première valeur";
@tableau[1] = "deuxième valeur";
@tableau[2] = "troisième valeur";
```

Vous ne pourrez pas ajouter une quatrième valeur à ce tableau :

```
my @tableau[3];
@tableau[0] = "première valeur";
@tableau[1] = "deuxième valeur";
@tableau[2] = "troisième valeur";
@tableau[3] = "quatrième valeur";

Index 3 for dimension 1 out of range (must be 0..2)
```

3-2-2. Tableaux à plusieurs dimensions

Les tableaux vus précédemment ne sont qu'à une dimension. Heureusement, nous pouvons en Perl 6 déclarer des tableaux de dimensions multiples.

```
my @multi-tab[3;2];
```

Ce tableau a deux dimensions. La première dimension peut contenir un maximum de trois valeurs et la seconde un maximum de deux valeurs.

```
my @multi-tab[3;2];
@multi-tab[0;0] = 1;
@multi-tab[0;1] = "x";
@multi-tab[1;0] = 2;
```

```
@multi-tab[1;1] = "y";
@multi-tab[2;0] = 3;
@multi-tab[2;1] = "z";
say @multi-tab
```

Pour la référence complète des tableaux : **http://doc.perl6.org/type/Array**.

3-3. Hachage

Un hachage (table de hachage/hash) est un ensemble de paires clef/valeur.

```
my %capitales = ('Angleterre','Londres','France','Paris');
say %capitales;
```

Une autre façon succincte de remplir le hachage :

```
my %capitales = (Angleterre => 'Londres', France => 'Paris');
say %capitales;
```

Voici quelques-unes des méthodes qui peuvent être appelées sur les hachages :

Script
```
my %capitales = (Angleterre => 'Londres', Allemagne => 'Berlin');
%capitales.push: (France => 'Paris');
say %capitales;
say %capitales.kv;
say %capitales.keys;
say %capitales.values;
say "La capitale de la France est: " ~ %capitales<France>;
```

Sortie
```
{Allemagne => Berlin, Angleterre => Londres, France => Paris}
(France Paris Allemagne Berlin Angleterre Londres)
(France Allemagne Angleterre)
(Paris Berlin Londres)
La capitale de la France est: Paris
```

Explication

`.push: (clef => 'valeur')` ajoute une nouvelle paire clef/valeur.

`.kv` renvoie la liste contenant toutes les clefs et valeurs.

`.keys` renvoie une liste des clefs.

`.values` renvoie une liste des valeurs.
On peut accéder à la valeur particulière d'un hachage en spécifiant sa clef, comme suit : `%hachage<clef>`

3-4. Types

Dans les exemples précédents, on n'a pas précisé quel type de valeurs les variables peuvent contenir.

`.WHAT` retournera le type de la valeur contenue dans la variable.

```
my $var = 'Texte';
say $var;
say $var.WHAT;

$var = 123;
say $var;
say $var.WHAT;
```

Comme vous pouvez le voir dans l'exemple ci-dessus, le type de valeur contenu dans `$var` était (`Str`) et puis (`Int`).

Ce style de programmation est appelé le typage dynamique. Dynamique dans le sens que les variables peuvent contenir des valeurs de tout type.

Maintenant, essayez d'exécuter l'exemple ci-dessous : Remarquez `Int` avant le nom de la variable.

```
my Int $var = 'Texte';
say $var;
say $var.WHAT;
```

Il va échouer et retourner ce message d'erreur: `Type check failed in assignment to $var; expected Int but got Str`.

Ce qui est arrivé est que nous avons précisé au préalable que la variable doit être de type (`Int`). Quand nous avons essayé de lui affecter un (`Str`), le programme a échoué.

Ce style de programmation est appelé le typage statique. Statique dans le sens que les types de variables sont définis avant l'affectation et ne peuvent pas changer.

Perl 6 possède un **typage graduel,** les deux typages, **statique** et **dynamique**, peuvent être utilisés.

Voici une liste des types les plus couramment utilisés.

Les deux premiers ne seront probablement jamais utilisés, mais ils sont répertoriés à titre informatif.

Type	Description	Exemple	Résultat
Mu	La racine de la hiérarchie de types		
Any	Classe de base par défaut pour les nouvelles classes et pour la plupart des classes intégrées		
Cool	Valeur qui peut être considérée comme une chaîne ou un nombre interchangeable	my Cool $var = 31; say $var.flip; say $var * 2;	13 62
Str	Chaîne de caractères	my Str $var = "NEON"; say $var.flip;	NOEN
Int	Entier (précision arbitraire)	7 + 7	14
Rat	Nombre rationnel (précision limitée)	0.1 + 0.2	0.3
Bool	Booléen	!True	False

3-5. Introspection

L'introspection est le processus d'obtention d'informations sur les propriétés d'un objet comme son type.
Dans l'exemple précédent, nous avons utilisé `.WHAT` pour connaître le type de la variable.

```
my Int $var;
say $var.WHAT;     # (Int)
my $var2;
say $var2.WHAT;    # (Any)
$var2 = 1;
say $var2.WHAT;    # (Int)
$var2 = "Hello";
say $var2.WHAT;    # (Str)
$var2 = True;
say $var2.WHAT;    # (Bool)
$var2 = Nil;
say $var2.WHAT;    # (Any)
```

Le type d'une variable contenant une valeur est corrélé à sa valeur.
Le type d'une variable vide fortement déclarée est le type avec lequel elle a été déclarée.
Le type d'une variable vide qui n'a pas été déclarée fortement est (`Any`).
Pour vider la valeur d'une variable, vous pouvez lui affecter `Nil`.

3-5-1. Portée

Avant d'utiliser une variable pour la première fois, elle doit être déclarée.

Plusieurs déclarateurs peuvent être utilisés dans Perl 6, `my` est ce que nous avons utilisé jusqu'ici.

```
my $var=1;
```

Le déclarateur `my` donne à la variable une portée **lexicale**. En d'autres termes, la variable ne sera accessible que dans le bloc où elle a été déclarée.

Un bloc en Perl 6 est délimité par `{ }`. Si aucun bloc n'est trouvé, la variable sera disponible dans l'ensemble du script (on dit alors parfois qu'elle est globale au script).

```
{
  my Str $var = 'Texte';
  say $var; #accessible
}
say $var; #inaccessible, renvoie une erreur
```

Comme une variable est uniquement accessible dans le bloc où elle est définie, le même nom de variable peut être redéfini dans un autre bloc.

```
{
  my Str $var = 'Texte';
  say $var;
}
my Int $var = 123;
say $var;
```

3-6. Affecter vs. Lier

Nous avons vu dans les exemples précédents comment **affecter** des valeurs aux variables. L'**affectation** est faite en utilisant l'opérateur `=`

```
my Int $var = 123;
say $var;
```

Nous pouvons modifier la valeur attribuée à une variable :

Affecter

```
my Int $var = 123;
say $var;
$var = 999;
say $var;
```

Sortie

```
123
999
```

D'autre part, nous ne pouvons pas changer la valeur **liée** à une variable. Le **lien** est établi en utilisant l'opérateur `:=`

Lier

```
my Int $var := 123;
say $var;
$var = 999;
say $var;
```

Sortie

```
123
Cannot assign to an immutable value
```

Une variable peut être également liée à une autre :

```
my $a;
my $b := $a;
$a = 7;
say $b;
```

Un lien ne peut être créé que lors de l'initialisation de la variable liée, et ne peut plus être modifié ensuite. Mais la valeur de la variable liée peut néanmoins changer si la valeur de la variable « maîtresse » à laquelle elle est liée change.

4. Fonctions normales et fonctions mutatrices

Il est important de différencier entre les fonctions normales et les fonctions mutatrices. Les fonctions normales ne changent pas l'état initial de l'objet. Les fonctions mutatrices modifient l'état de l'objet.

Script

```
1.
2.
3.
4.
5.
```

```
                                                                    6.
                                                                    7.
                                                                    8.
                                                                    9.
                                                                   10.
my @numeros = [7,2,4,9,11,3];

@numeros.push(99);
say @numeros;      #1

say @numeros.sort; #2
say @numeros;      #3

@numeros.=sort;
say @numeros;      #4
```

Sortie

```
 [7 2 4 9 11 3 99] #1
(2 3 4 7 9 11 99) #2
[7 2 4 9 11 3 99] #3
[2 3 4 7 9 11 99] #4
```

Explication

`.push` est une fonction mutatrice, elle change l'état du tableau (#1).

`.sort` est une fonction normale, elle retourne un tableau trié, mais ne modifie pas l'état initial du tableau :

- (#2) démontre le retour d'un tableau trié ;
- (#3) démontre que le tableau initial reste non modifié.

Afin de forcer une fonction normale à agir comme une fonction mutatrice, nous pouvons utiliser `.=` a la place de `.` (#4) (Ligne 9 du script).

5. Structures conditionnelles et boucles

Perl 6 possède une multitude de structures conditionnelles et structures de boucles.

5-1. if

Le code ne s'exécute que si la condition a été remplie.

```
my $âge = 19;

if $âge > 18 {
  say 'Bienvenue'
}
```

En Perl 6, nous pouvons inverser le code et la condition. Même si le code et la condition ont été inversés, la condition est toujours évaluée en premier.

```
my $âge = 19;

say 'Bienvenue' if $âge > 18;
```

Si la condition n'est pas remplie, nous pouvons toujours préciser des blocs d'exécution alternatifs en utilisant :

- `else`
- `elsif`

```
#exécuter le même code pour différentes valeurs de la variable
my $nombre-de-places = 9;

if $nombre-de-places <= 5 {
  say 'Je suis une berline'
} elsif $nombre-de-places <= 7 {
  say 'Je suis un monospace'
} else {
  say 'Je suis un van'
}
```

5-2. unless

La version négative d'un `if` peut être écrite en utilisant `unless`.

Le code suivant :

```
my $chaussures-propres = False;

if not $chaussures-propres {
  say 'Nettoyez vos chaussures'
}
```

peut aussi être écrit ainsi :

```
my $chaussures-propres = False;

unless $chaussures-propres {
  say 'Nettoyez vos chaussures'
}
```

La négation en Perl 6 est faite en utilisant `!` ou `not`.

`unless (condition)` est utilisé à la place de `if not (condition)`.

`unless` ne peut pas avoir une clause `else`.

5-3. with

`with` fonctionne comme `if`, mais vérifie si la variable est définie.

```
my Int $var=1;

with $var {
  say 'Bonjour'
}
```

Si vous exécutez le code sans attribuer une valeur à la variable, il ne se passera rien.

```
my Int $var;
```

```
with $var {
  say 'Bonjour'
}
```

`without` est la version négative de `with`. Vous devriez être capable de relier le concept à `unless`.

Si la première condition `with` n'est pas remplie, un autre chemin peut être spécifié en utilisant `orwith`.
`with` et `orwith` peuvent être comparés à `if` et `elsif`.

5-4. for

La boucle for itère sur plusieurs valeurs.

```
my @tableau = 1,2,3;

for @tableau -> $element {
  say $element*100
}
```

Notez que nous avons créé une variable d'itération `$element` afin d'effectuer l'opération `*100` sur chaque élément du tableau. Dans ce genre de construction, la variable d'itération `$element` est autodéclarée et ne doit donc pas être précédée par le déclarateur `my`.

5-5. given

`given` est l'équivalent Perl 6 de l'instruction `switch` dans d'autres langages.

```
my $var = 42;

given $var {
    when 0..50 { say 'Plus petit que 50'}
    when Int { say "est un Int" }
    when 42  { say 42 }
    default  { say "heu?" }
}
```

Si l'une des conditions est satisfaite, le processus d'appariement s'arrête (les autres conditions ne seront pas testées). Le code ci-dessus n'affichera donc que « Plus petit que 50 ».

Si l'on préfère tester aussi les conditions suivantes, `proceed` instruira Perl 6 à poursuivre l'appariement, même après un appariement réussi.

```
my $var = 42;

given $var {
    when 0..50 { say 'Plus petit que 50';proceed}
    when Int { say "est un Int";proceed}
    when 42  { say 42 }
    default  { say "huh?" }
}
```

5-6. loop

`loop` est une autre façon d'écrire une boucle `for`.

En fait `loop` s'écrit comme le sont les boucles `for` dans les langages de programmation appartenant à la famille C.

Perl 6 appartient à la famille C.

```
loop (my $i=0; $i < 5; $i++) {
    say "Le nombre actuel est $i"
}
```

6. Entrées/Sorties

En Perl 6, deux des interfaces ***entrée/sortie*** les plus communes sont le ***Terminal*** et les ***Fichiers***.

6-1. E/S Basic en utilisant le terminal

6-1-1. say

`say` écrit sur la sortie standard (en général, l'écran). Il ajoute un caractère de fin ligne à la fin. Autrement dit, le code suivant :

```
say 'Bonjour Madame.';
say 'Bonjour Monsieur.';
```

sera écrit sur deux lignes distinctes.

6-1-2. print

`print` fonctionne comme `say`, mais sans ajouter de caractère de fin ligne.

Essayez de remplacer `say` avec `print` et de comparer les deux résultats.

6-1-2-1. get

`get` est utilisé pour capturer l'entrée du terminal.

```
my $nom;

say "Salut quel est ton nom?";
$nom=get;

say "Cher $nom bienvenue à Perl 6";
```

Lorsque le code ci-dessus est lancé, le terminal attendra que vous saisissiez votre nom. Par la suite, il vous accueillera.

6-1-3. prompt

`prompt` est une combinaison de `print` et `get`.

L'exemple ci-dessus peut être écrit comme ceci :

```
my $nom = prompt("Salut quel est ton nom? ");

say "Cher $nom bienvenue à Perl 6";
```

6-2. Exécution de commandes Shell

Deux routines peuvent être utilisées pour exécuter des commandes shell :

- `run` : exécute une commande externe sans impliquer le shell ;
- `shell` : exécute une commande via le shell. Tous les métacaractères sont interprétés par le shell, y compris les tubes (*pipes*), les redirections, les variables d'environnement, etc.

Voici un exemple sous Linux, Unix ou OS X :

```
my $nom = 'Neo';
run 'echo', "salut $nom";
shell "ls";
```

Et un exemple sous Windows :

```
shell "dir";
```

`echo` et `ls` sont des mots-clefs communs des shells Unix ou Linux. `echo` imprime le texte sur le terminal (l'équivalent de `say` en Perl 6). `ls` liste tous les fichiers et dossiers dans le répertoire courant sous Linux et `dir` fait la même chose sous Windows.

6-3. E/S Fichier

6-3-1. slurp

`slurp` est utilisé pour lire les données d'un fichier.

Créez un fichier texte avec le contenu suivant :

datafile.txt

```
John 9
Johnnie 7
Jane 8
Joanna 7
```

```
my $data = slurp "datafile.txt";
say $data;
```

6-3-2. spurt

`spurt` est utilisé pour écrire des données sur un fichier.

```
my $newdata = "New scores:
Paul 10
Paulie 9
Paulo 11";
```

```
spurt "newdatafile.txt", $newdata;
```

Après avoir exécuté le code ci-dessus, un nouveau fichier nommé ***newdatafile.txt*** sera créé. Il contiendra les nouveaux scores.

6-4. Travailler avec les fichiers et répertoires

Perl 6 peut lister le contenu d'un répertoire sans exécuter des commandes shell (en utilisant `ls`) comme nous l'avons vu dans un exemple précédent.

```
say dir;               #Liste les fichiers et dossiers dans le
répertoire courant
say dir "Documents"; #Liste les fichiers et dossiers dans le
répertoire spécifié
my @répertoire = dir; # Récupère les fichiers dans un tableau
```

De plus, vous pouvez créer de nouveaux dossiers et les supprimer.

```
mkdir "newfolder";
rmdir "newfolder";
```

`mkdir` crée un nouveau répertoire.
`rmdir` supprime un répertoire vide. Renvoie une erreur s'il n'est pas vide.

Vous pouvez également vérifier si le chemin d'accès spécifié existe, si c'est un fichier ou un répertoire.

Dans le répertoire où vous allez exécuter le script ci-dessous, créez un dossier vide `folder123` et un fichier Perl 6 vide `script123.pl6`

```
say "script123.pl6".IO.e;
say "folder123".IO.e;

say "script123.pl6".IO.d;
say "folder123".IO.d;

say "script123.pl6".IO.f;
say "folder123".IO.f;
```

La méthode `IO` sert à transformer la chaîne de caractères « script123 » en un objet de type `IO::Path`. Les méthodes « `e` », « `f` » et « `d` » de tests de fichiers ne peuvent être invoquées que sur des objets de type `IO::Path`, d'où la nécessité de coercition préalable de la chaîne de caractères en un objet de ce type.

`IO.e` vérifie si le répertoire/fichier existe.
`IO.f` vérifie si c'est un fichier.
`IO.d` vérifie si c'est un dossier.

Les utilisateurs Windows peuvent utiliser / ou \\ comme séparateur entre les dossiers :

`C:\\rakudo\\bin`

`C:/rakudo/bin`

Pour plus d'informations sur les E/S, voir **http://doc.perl6.org/type/IO**.

7. Routines

7-1. Définition

Les **routines** ou **subroutines** ou **subs** sont un moyen de conditionnement d'un ensemble de fonctionnalités.

Une routine est définie avec le mot-clef `sub`. Après leur définition, elles peuvent être appelées par leur nom.
Examinez l'exemple ci-dessous :

```
sub salut-alien {
  say "Bonjour Terriens";
}

salut-alien;
```

L'exemple précédent présente une routine qui ne nécessite aucun argument.

7-2. Signature

Beaucoup de routines requièrent des données en entrée pour fonctionner. Ces données sont fournies par des **arguments**.
La **signature** est le nombre et le type d'arguments que la routine accepte.

La routine ci-dessous accepte une chaîne de caractères pour argument :

```
sub dis-bonjour (Str $nom) {
    say "Bonjour " ~ $nom ~ "!!!!"
}
dis-bonjour "Paul";
dis-bonjour "Paula";
```

7-3. Multiroutines

Il est possible de définir plusieurs routines ayant le même nom, mais des signatures différentes. Lorsque la routine est appelée, l'environnement d'exécution décidera quelle version utiliser en fonction du nombre et du type des arguments fournis. Ce type de routines est défini de la même manière que les routines normales sauf que nous utilisons le mot-clef `multi` à la place de `sub`.

```
multi salut($nom) {
    say "Bonne Journée $nom";
}
multi salut($nom, $titre) {
    say "Bonne Journée $titre $nom";
}

salut "Gaspard";
salut "Josiane","Mme.";
```

7-4. Arguments optionnels et par défaut

Si une routine est définie comme acceptant un argument, et nous l'appelons sans fournir l'argument requis, la routine va échouer.

Cependant, Perl 6 nous offre la possibilité de définir des routines avec des :

- arguments optionnels ;
- arguments par défaut.

Les arguments optionnels sont définis en ajoutant `?` après le nom de l'argument.

```
sub dis-bonjour($nom?) {
  with $nom { say "Bonjour " ~ $nom }
  else { say "Bonjour être humain" }
}
dis-bonjour;
dis-bonjour("Laura");
```

Si l'utilisateur ne fournit pas un argument, la routine peut fournir une valeur par défaut. Cela se fait par l'attribution d'une valeur à l'argument durant la définition de la routine.

```
sub dis-bonjour($nom="Raoul") {
  say "Bonjour " ~ $nom;
}
dis-bonjour;
dis-bonjour("Laura");
```

8. Programmation fonctionnelle

Ce chapitre traitera des fonctionnalités facilitant la programmation fonctionnelle.

8-1. Les fonctions sont des entités de première classe

Les fonctions/routines sont des entités de première classe :

- elles peuvent être passées comme un argument ;
- elles peuvent être renvoyées par une fonction ;
- on peut les affecter à une variable.

Un bon exemple pour vérifier ce concept est la fonction `map`. `map` est une **fonction d'ordre supérieur**, elle accepte une autre fonction comme argument.

Script
```
my @tableau = <1 2 3 4 5>;
sub carré($x) {
  $x ** 2
}
say map(&carré,@tableau);
```

Sortie
```
(1 4 9 16 25)
```

Explication

Nous avons défini une routine appelée `carré`, qui met à la puissance 2 l'argument qui lui est passé.
Ensuite, nous avons utilisé `map`, une fonction d'ordre supérieur en lui passant deux arguments, une routine et un tableau.
Le résultat est une liste de tous les éléments du tableau mis à la puissance 2.

Notez que quand une routine est passée comme argument, nous la préfixons avec `&`.

8-2. Fermeture

Tous les objets code en Perl 6 sont des fermetures, ce qui implique qu'ils peuvent référencer des variables lexicales d'une portée externe.

8-3. Fonctions anonymes

Une **fonction anonyme** est également appelée **lambda**.
Une fonction anonyme n'est pas liée à un identifiant (elle n'a pas de nom).

Réécrivons l'exemple de `map` avec une fonction anonyme

```
my @tableau = <1 2 3 4 5>;
say map(-> $x {$x ** 2},@tableau);
```

Notez qu'au lieu de définir une routine et de la passer en argument à `map`, nous la définissons directement à l'intérieur de `map`.
La routine anonyme `-> $x {$x ** 2}` n'a pas de nom et ne peut donc pas être appelée.

En dialecte Perl 6, nous l'appelons un **bloc pointu** (*pointy block*).

Un bloc pointu peut aussi être utilisé pour assigner des fonctions à des variables :

```
my $carré = -> $x {
  $x ** 2
}
say $carré(9);
```

8-4. Enchaînement

En Perl 6, les méthodes peuvent être enchaînées, vous n'avez plus à passer le résultat d'une méthode comme argument à une autre.

Supposons qu'on vous donne un tableau de valeurs. On vous demande de retourner les valeurs uniques de ce tableau en ordre décroissant.

Vous pouvez résoudre ce problème en écrivant quelque chose comme ceci :

```
my @tableau = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
my @tableau-final = reverse(sort(unique(@tableau)));
say @tableau-final;
```

Nous appelons d'abord la fonction `unique` sur `@tableau` puis nous passons le résultat comme argument à `sort` et ensuite passons le résultat à `reverse`.

L'exemple ci-dessus peut aussi être écrit comme suit, en prenant avantage de **l'enchaînement des méthodes** :

```
my @tableau = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
my @tableau-final = @tableau.unique.sort.reverse;
say @tableau-final;
```

Vous pouvez constater qu'enchaîner les méthodes est ***plus agréable à l'œil et au cerveau***.

8-5. Opérateur feed

L'opérateur **feed**, appelé `Pipe` dans d'autres langages fonctionnels, donne une meilleure vue de l'enchaînement de méthodes.

Feed vers l'avant

```
my @tableau = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
@tableau ==> unique()
         ==> sort()
         ==> reverse()
         ==> my @tableau-final;
say @tableau-final;
```

Explication

```
commence avec `@tableau` puis renvoie la liste des éléments uniques
                         puis effectue un tri
                         puis l'inverse
                         puis stocke le résultat dans @tableau-final
```

Comme vous le voyez, le flux des appels de méthodes se fait de haut en bas.

Feed vers l'arrière

```
my @tableau = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
my @tableau-final-v2 <== reverse()
                     <== sort()
                     <== unique()
                     <== @tableau;
say @tableau-final-v2;
```

Explication

Le feed vers l'arrière est comme celui vers l'avant, mais se fait à rebours. Le flux des appels de méthodes se fait de bas en haut.

8-6. Hyperopérateur

L'**hyperpérateur** `>>.` invoque une méthode sur tous les éléments d'une liste et renvoie une liste des résultats.

```
my @tableau = <0 1 2 3 4 5 6 7 8 9 10>;
sub est-pair($var) { $var %% 2 };

say @tableau>>.is-prime;
say @tableau>>.&est-pair;
```

En utilisant l'hyperopérateur, nous pouvons appeler des méthodes déjà définies dans Perl 6, par exemple `is-prime` qui nous indique si un nombre est premier ou pas. Nous pouvons également définir de nouvelles routines et les appeler en utilisant l'hyperopérateur. En ce cas, il faut préfixer la méthode avec `&`. Ex. : `&est-pair`.

Cette façon de faire est très pratique, car elle nous évite d'écrire une boucle `for` d'itération sur chaque valeur.

8-7. Jonctions

Une **jonction** est une superposition logique des valeurs.

Dans l'exemple ci-dessous, `1|2|3` est une jonction.

```
my $var = 2;
if $var == 1|2|3 {
  say "La variable est soit 1 ou 2 ou 3";
}
```

L'utilisation de jonctions déclenche généralement l'**autothreading** ; l'opération est effectuée pour chaque élément de la jonction, les résultats sont combinés en une seule jonction et renvoyés.

8-8. Listes paresseuses

Une **liste paresseuse** est une liste dont l'évaluation peut être retardée. L'évaluation paresseuse diffère l'évaluation d'une expression jusqu'au moment où celle-ci est nécessaire, et évite ainsi la répétition des évaluations en stockant les résultats dans une table de correspondance.

Les avantages, parmi d'autres, sont les suivants :

- un gain de performance évitant les calculs inutiles ;
- la possibilité de construire des structures de données potentiellement infinies ;
- la possibilité de définir une structure de contrôle.

Pour construire une liste paresseuse, on utilise l'opérateur infixé `...` Une liste paresseuse possède **un ou des éléments initiaux**, **un générateur**, et un **élément final**.

Liste paresseuse simple

```
my $lazylist = (1 ... 10);
say $lazylist;
```

L'élément initial est 1, et l'élément final est 10. Aucun générateur n'a été défini donc le générateur par défaut se fait par succession (+1). Autrement dit, cette liste paresseuse retournera (à la demande) les éléments suivants : (1, 2, 3, 4, 5, 6, 7, 8, 9, 10).

Liste paresseuse infinie

```
my $lazylist = (1 ... Inf);
say $lazylist;
```

Cette liste retournera (à la demande) les entiers entre 1 et l'infini, c'est-à-dire tous les entiers.

Liste paresseuse utilisant un générateur déduit

```
my $lazylist = (0,2 ... 10);
say $lazylist;
```

Les éléments initiaux sont 0 et 2, et l'élément final est 10. Aucun générateur n'est défini, mais en utilisant les éléments initiaux, Perl 6 déduira que le générateur est (+2). Cette liste paresseuse retournera (à la demande) les éléments suivants : (0, 2, 4, 6, 8, 10).

Liste paresseuse utilisant un générateur défini

```
my $lazylist = (0, { $_ + 3 } ... 12);
say $lazylist;
```

Dans cet exemple, nous définissons explicitement un générateur mis entre `{ }`
Cette liste paresseuse retournera (à la demande) les éléments suivants : (0, 3, 6, 9, 12).

Si vous utilisez un générateur explicite, l'élément final doit être une valeur que le générateur puisse retourner.
Si nous reproduisons l'exemple ci-dessus avec un élément final égal à 10 au lieu de 12, il n'y aura pas de fin. Le générateur ***saute par dessus*** l'élément final.

Vous pouvez sinon remplacer `0 ... 10` par `0 ...^ * > 10`
Ce qui se lit comme : de 0 jusqu'à la première valeur supérieure à 10 exclue.

Ceci ne stoppera pas le générateur

```
my $lazylist = (0, { $_ + 3 } ... 10);
say $lazylist;
```

Ceci stoppera le générateur

```
my $lazylist = (0, { $_ + 3 } ...^ * > 10);
say $lazylist;
```

9. Classes et Objets

9-1. Introduction

La Programmation ***Orientée Objet*** est l'un des paradigmes les plus utilisés de nos jours.
Un **objet** est une collection de variables et de routines empaquetées ensemble.
Les variables sont appelées des **attributs** et les routines des **méthodes**.
Les attributs définissent l'**état** et les méthodes le **comportement** d'un objet.

Une **classe** définit la structure d'une collection d'**objets**.

Pour comprendre cette relation, examinez l'exemple ci-dessous :

Il y a 4 personnes présentes dans une pièce	**objets** ⇒ 4 personnes
Ces 4 personnes sont des êtres humains	**classe** ⇒ Humain
Ils ont des noms, âges, sexes et nationalités différents	**attributs** ⇒ nom, âge, sexe, nationalité

Dans le jargon ***orienté objet,*** nous disons que les objets sont des **instances** d'une classe.

Voyez le script ci-dessous :

```
class Humain {
  has $nom;
  has $age;
  has $sexe;
  has $nationalité;
}

my $françois;
$françois = Humain.new(nom => 'François', age => 23, sexe => 'M',
nationalité => 'Sarthoise');
say $françois;
```

Le mot-clef `class` est utilisé pour définir une classe.
Le mot-clef `has` est utilisé pour définir un attribut d'une classe.
La méthode `.new()` est appelée un **constructeur**. Elle crée l'objet comme une instance de la classe sur laquelle elle a été appelée.

Dans le script ci-dessus, la nouvelle variable `$françois` contient une référence vers une nouvelle instance de « Humain » définie par `Humain.new()`.
Les arguments passés à la méthode `.new()` sont utilisés pour initialiser les attributs de l'objet.

Une classe peut se voir donner une ***portée lexicale*** en utilisant `my` :

```
my class Humain {

}
```