Sébastien Aperghis-Tramoni
Philippe Bruhat
Damien Krotkine
Jérôme Quelin

LE GUIDE DE SURVIE

Perl moderne

L'ESSENTIEL DES PRATIQUES ACTUELLES

PEARSON

Perl moderne

Sébastien Aperghis-Tramoni
Damien Krotkine
Jérôme Quelin

avec la contribution
de Philippe Bruhat

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél : 01 72 74 90 00
www.pearson.fr

Réalisation PAO : euklides.fr
Collaboration éditoriale : Jean-Philippe Moreux

ISBN : 978-2-7440-4164-8

Table des matières

Les auteurs XVI
Avant-propos XVII

1 Démarrer avec Perl 1
Vérifier la version de Perl 1
Exécuter les exemples 2
Exécuter `perl` sur un fichier 3
Éditer les programmes 4
Installer Perl sous Linux 5
Installer Perl sous Windows 6
Installer Perl sous Mac OS X 6

2 Installer un module Perl 7
Chercher un module avec `cpan` 9
Installer un module avec `cpan` 10
Mettre à jour les modules 10
Installer un module avec `cpanm` 11

Partie I – Langage et structures de données

3 Éléments du langage 13
Exécuter `perl` en ligne de commande 13
Exécuter `perl` sur un fichier 15
Créer un fichier exécutable 15
Exécuter en mode déboggage 15
Règles générales du langage 17
Les types de données 18
Initialiser une variable scalaire 19

Déclarer une variable scalaire 19
Afficher un scalaire 20
Créer une chaîne de caractères 20
La notion de contexte 22
Travailler sur les nombres 24
Travailler sur les chaînes 25
Tester si un scalaire est défini 26
Déclarer une fonction 26
Passer des paramètres 27
Renvoyer une valeur de retour 28
Utiliser des variables dans une fonction 29
Les opérateurs de test 29
Tester une expression 31
Effectuer une opération conditionnelle 32
Effectuer une opération si un test est faux 33
Tester négativement une expression 34
Effectuer des tests avec `and` et `or` 34
Boucler sur les éléments d'une liste 35
Boucler tant qu'un test est vrai 36
Créer une référence sur scalaire 37
Déréférencer une référence sur scalaire 38
Accéder à une variable référencée 38
Passer un paramètre par référence 39
Utiliser des références sur fonctions 40
Récupérer les arguments de la ligne de commande 41
Exécuter des commandes système 42
Terminer abruptement un programme 42
Créer un module Perl 43
Utiliser un module Perl 44

4 Structures de données 47
Créer une liste 47
Créer une liste avec un intervalle 48
Créer une liste de mots 49

Créer un tableau à une dimension 50
Accéder aux éléments d'un tableau 51
Affecter un élément d'un tableau 52
Obtenir le premier élément d'un tableau 53
Obtenir le dernier élément d'un tableau 53
Obtenir la taille d'un tableau 54
Assigner un élément en dehors du tableau 55
Tester les éléments d'un tableau 55
Manipuler la fin d'un tableau 56
Manipuler le début d'un tableau 58
Manipuler le milieu d'un tableau 58
Supprimer un élément d'un tableau 60
Inverser une liste ou un tableau 60
Aplatir listes et tableaux 61
Manipuler une tranche de tableau 62
Boucler sur les éléments d'un tableau 65
Créer un tableau à plusieurs dimensions 67
Référencer un tableau 67
Déréférencer un tableau 68
Créer des références dans un tableau 69
Accéder à un tableau de tableaux 70
Modifier un tableau de tableaux 71
Dumper un tableau 71
Utiliser les tables de hachage 74
Créer une table de hachage 75
Accéder aux éléments d'une table de hachage 76
Supprimer un élément d'une table de hachage 78
Tester l'existence et la définition d'un élément 78
Utiliser des tranches de hashs 79
Obtenir la liste des clés 80
Obtenir la liste des valeurs d'une table de hachage 81
Dumper un hash 82
Boucler sur un hash avec `foreach` 82
Boucler sur un hash avec `each` 83

Référencer un hash 84
Déréférencer un hash 84
Créer des structures hybrides 85
Transformer tous les éléments d'un tableau ou d'une liste avec `map` 86
Filtrer un tableau ou une liste avec `grep` 88
Renvoyer le premier élément d'une liste avec `List::Util` 89
Trouver le plus grand élément avec `List::Util` 90
Trouver le plus petit élément avec `List::Util` 90
Réduire une liste avec `List::Util` 91
Mélanger une liste avec `List::Util` 92
Faire la somme d'une liste avec `List::Util` 92
Savoir si un élément vérifie un test avec `List::MoreUtils` 92
Savoir si aucun élément ne vérifie un test avec `List::MoreUtils` 93
Appliquer du code sur deux tableaux avec `List::MoreUtils` 93
Tricoter un tableau avec `List::MoreUtils` 93
Enlever les doublons avec `List::MoreUtils` 94

5 Expressions régulières 95
Effectuer une recherche 96
Rechercher et remplacer 97
Stocker une expression régulière 98
Rechercher sans prendre en compte la casse 99
Rechercher dans une chaîne multiligne 100
Rechercher dans une chaîne simple 100
Neutraliser les caractères espace 101
Contrôler la recherche globale de correspondances 101
Distinguer caractères normaux et métacaractères 102
Établir une correspondance parmi plusieurs caractères 104
Ancrer une expression régulière en début de ligne 106
Ancrer une expression régulière en fin de ligne 107
Utiliser une ancre de début de chaîne 108

Utiliser une ancre de fin de chaîne 108
Utiliser une ancre de frontière de mot 108
Utiliser une ancre par préfixe de recherche 109
Utiliser une ancre de correspondance globale 109
Quantifieur * 111
Quantifieur + 112
Quantifieur ? 112
Quantifieur {n} 113
Quantifieur {n,m} 113
Quantifieurs non avides 114
Quantifieurs possessifs 115
Capturer du texte avec les groupes capturants 116
Grouper sans capturer avec les groupes non capturants 119
Définir des alternatives 121
Découper une chaîne avec `split` 122
Utiliser `Regexp::Common` 124
Utiliser `Regexp::Assemble` 126
Utiliser `Text::Match::FastAlternatives` 128
Utiliser `YAPE::Regex::Explain` 128

Partie II – Objet moderne

6 Concepts objet en Perl 131
Créer un objet 132
Connaître la classe d'un objet 132
Appeler une méthode 133
Définir une méthode 133
Définir un constructeur 135

7 Moose 137
Déclarer une classe 137
Déclarer un attribut 138
Accéder aux objets 140
Modifier le nom des accesseurs 141

Méthodes de prédicat et de suppression 143
Rendre un attribut obligatoire 144
Vérifier le type d'un attribut 145
Donner une valeur par défaut 145
Construire un attribut 147
Rendre un attribut paresseux 148
Spécifier un déclencheur 150
Déréférencer un attribut 151
Affaiblir un attribut référence 152
Chaîner les attributs 153
Simplifier les déclarations d'attributs 153
Étendre une classe parente 154
Surcharger une méthode 155
Modifier des attributs hérités 156
Créer un rôle 156
Consommer un rôle 157
Requérir une méthode 158
Construire et détruire des objets 159
Modifier les paramètres du constructeur 159
Interagir avec un objet nouvellement créé 160
Interagir lors de la destruction d'un objet 161

8 Le typage dans Moose 163
Utiliser les types de base 163
Créer une bibliothèque de types personnalisés 165
Définir un sous-type 165
Définir un nouveau type 166
Définir une énumération 167
Définir une union de types 167
Transtyper une valeur 168

9 Moose et les méthodes 171
Modifier des méthodes 171
Intercaler un prétraitement 172

Intercaler un post-traitement 173
S'intercaler autour d'une méthode 174
Modifier plusieurs méthodes 175
Appeler la méthode parente 176
Augmenter une méthode 177
Déléguer une méthode à un attribut 179
Déléguer des méthodes à une structure Perl 180

Partie III – Manipulation de données

10 Fichiers et répertoires 183
Ouvrir des fichiers 183
Utiliser un descripteur de fichier en lecture 185
Utiliser un descripteur de fichier en écriture 187
Fermer un descripteur de fichier 188
Manipuler des chemins avec `Path::Class` 189
Pointer un fichier ou un répertoire 190
Pointer un objet relatif 191
Pointer un objet parent 191
Obtenir des informations 192
Créer ou supprimer un répertoire 193
Lister un répertoire 193
Ouvrir un fichier 194
Supprimer un fichier 196
Parcourir un répertoire récursivement 196
Créer un fichier temporaire 197
Créer un répertoire temporaire 198
Identifier les répertoires personnels 199
Connaître le répertoire courant 201
Changer de répertoire 201

11 Bases de données SQL 203
Se connecter 205
Tester la connexion 208

Se déconnecter 209
Préparer une requête SQL 209
Lier une requête SQL 211
Exécuter une requête SQL 211
Récupérer les données de retour d'une requête SQL 212
Combiner les étapes d'exécution d'une requête SQL 214
Gérer les erreurs 216
Tracer l'exécution 217
Profiler l'exécution 219

12 Abstraction du SQL, ORM et bases non-SQL 223
Utiliser `Data::Phrasebook::SQL` 223
ORM avec `DBIx::Class` 232
Créer un schéma `DBIx::Class` 234
Utiliser un schéma `DBIx::Class` 237
Stocker des objets avec KiokuDB 238
Se connecter à une base `KiokuDB` 239
Stocker et récupérer des objets 240
Administrer une base KiokuDB 242
Utiliser une base orientée paires de clé-valeur 243
Utiliser une base orientée documents 244

13 Dates et heures 247
Utiliser le module `Date::Parse` 249
Lire une date avec `Date::Parse` 250
Interpréter une date avec `Date::Parse` 250
Changer la langue avec `Date::Language` 251
Gérer les intervalles de temps avec `Time::Duration` 252
Interpréter une durée avec `Time::Duration` 253
Obtenir la durée à partir de maintenant 254
Réduire l'affichage 254
Changer la langue avec `Time::Duration::fr` 255
Utiliser les modules `DateTime` 256
Construire une instance `DateTime` arbitraire 257
Choisir un fuseau horaire 258

Obtenir l'instant présent 258
Obtenir la date du jour 259
Obtenir l'année 259
Obtenir le mois 260
Obtenir le nom du mois 260
Obtenir le jour du mois 260
Obtenir le jour de la semaine 260
Obtenir le nom du jour 261
Obtenir l'heure 261
Obtenir les minutes 261
Obtenir les secondes 261
Obtenir les nanosecondes 262
Obtenir des durées de temps 263
Décaler une date dans le futur 264
Ajouter une durée 265
Décaler une date dans le passé 265
Soustraire une durée 266
Calculer un intervalle de temps 267
Générer une représentation textuelle d'une date 268
Interpréter une date 270

Partie IV – Formats structurés

14 XML 273
Charger un document XML avec `XML::LibXML` 275
Parcourir un arbre DOM 277
Utiliser XPath 280
Utiliser SAX 281
Créer un objet `XML::Twig` 285
Charger du contenu XML avec `XML::Twig` 286
Créer des handlers avec `XML::Twig` 287
Produire le contenu XML de sortie 289
Ignorer le contenu XML de sortie 289

Accéder au nom d'un élément 290
Changer le nom d'un élément 291
Obtenir le contenu texte d'un élément 292
Changer le contenu XML d'un élément 292
Interagir avec les attributs d'un élément 293
Interagir avec les éléments environnants 294
Effectuer un copier-coller 297

15 Sérialisation de données 301
Sérialiser avec `Data::Dumper` 301
Sérialiser avec `Storable` 307
Sérialiser avec `JSON` 310
Sérialiser avec `YAML` 316

16 Fichiers de configuration 319
Fichiers .INI 320

Partie V – Programmation événementielle

17 Principes généraux de POE 325
POE 326
Événements 327
Sessions 328
Le noyau POE 329

18 POE en pratique 331
Créer une session POE 331
Envoyer un événement 332
Passer des paramètres 333
Utiliser des variables privées 335
Communiquer avec une autre session 336
Envoyer un message différé 339
Envoyer un message à l'heure dite 340

Terminer le programme 341
Couper les longs traitements 341
Bannir les entrées-sorties bloquantes 345
Composants de haut niveau 346
Boîte à outils de niveau intermédiaire 347
Fonctions de bas niveau POE 348
Exemple d'utilisation : le composant DBI 348

19 POE distribué 357
Créer un serveur IKC 358
Créer un client IKC 359

Partie VI – Web

20 Analyse de documents HTML 363
Analyser avec les expressions régulières 364
Utiliser l'analyseur événementiel HTML::Parser 365
Instancier un analyseur HTML::Parser 365
Créer un gestionnaire d'événements 366
Lancer l'analyse HTML 368
Terminer l'analyse du contenu 369
Détecter un nouveau document 369
Détecter une balise 370
Détecter un commentaire 370
Détecter un début de balise 371
Détecter du texte brut 371
Détecter la fin d'une balise 372
Détecter la fin du document 372
Détecter des instructions de traitement 372
Capturer les autres événements 373
Extraire du texte d'un document 373
Produire une table des matières 374
Créer une instance HTML::TokeParser 377

Récupérer des *tokens* 378
Obtenir des balises 379
Obtenir du texte 380
Obtenir du texte nettoyé 381
Extraire le texte d'un document avec `HTML::Parser` 381
Produire une table des matières avec `HTML::Parser` 382
Analyse par arbre avec `HTML::TreeBuilder` 383
Créer un arbre 384
Rechercher un élément 385
Extraire du texte d'un document avec `HTML::TreeBuilder` 387
Produire une table des matières avec `HTML::TreeBuilder` 387
Extraire le titre d'un document avec `HTML::TreeBuilder` 388

21 HTTP et le Web 389
Adresses 390
Messages 391
Requêtes 392
Réponses 393

22 LWP 395
Utiliser `LWP::Simple` 395
Faire une requête GET sur une URL 395
Enregistrer le contenu de la réponse 396
Faire une requête HEAD sur une URL 396
Utiliser `LWP::UserAgent` 397
Créer un agent `LWP::UserAgent` 397
Gérer les réponses 398
Faire une requête GET sur une URL avec `LWP::UserAgent` 399
Enregistrer le contenu de la réponse 400
Faire une requête HEAD sur une URL avec `LWP::UserAgent` 401
Faire une requête POST sur une URL avec `LWP::UserAgent` 402

Envoyer des requêtes 403
Différences entre LWP::UserAgent et un « vrai » navigateur 403

23 Navigation complexe 407
Traiter les erreurs 407
Authentifier 408
Gérer les cookies 410
Créer un objet HTML::Form 411
Sélectionner et modifier des champs de formulaire 412
Valider un formulaire 413

24 WWW::Mechanize 415
Créer un objet WWW::Mechanize 415
Lancer une requête GET 417
Lancer une requête POST 417
Revenir en arrière 417
Recharger une page 418
Suivre des liens 418
Traiter les erreurs 420
Authentifier 420
Gérer les cookies 420
Gérer les formulaires 421
Sélectionner un formulaire par son rang 421
Sélectionner un formulaire par son nom 421
Sélectionner un formulaire par son identifiant 422
Sélectionner un formulaire par ses champs 422
Remplir le formulaire sélectionné 422
Valider le formulaire sélectionné 423
Sélectionner, remplir et valider un formulaire 423
Exemple d'application 424

A Tableau récapitulatif des opérateurs 429

Index 431

Les auteurs

Les auteurs, activement impliqués dans la communauté Perl française, sont membres de l'association *Les mongueurs de Perl* (**http://www.mongueurs.net**).

Sébastien Aperghis-Tramoni travaille actuellement chez Orange France comme ingénieur système et développeur Perl. Utilisateur de ce langage depuis 15 ans, il contribue régulièrement à Perl, au CPAN (une quarantaine de modules), et à d'autres logiciels libres. Il participe à de nombreuses conférences en France et en Europe depuis 2003. Il est l'auteur de plusieurs articles parus dans *GNU/Linux Magazine France*.

Damien Krotkine, ingénieur, a travaillé notamment chez Mandriva (éditeur Linux), et Venda (leader eCommerce). Auteur de sept modules CPAN, il est également coauteur de *Linux – le guide complet* (Micro Application), et auteur d'articles dans *GNU/Linux Magazine France*. Ancien développeur Gentoo Linux et ancien contributeur Mandriva, il participe régulièrement aux conférences French Perl Workshop et OSDC.fr.

Jérôme Quelin utilise le langage Perl depuis 1995. Il est l'auteur d'une quarantaine de modules sur CPAN et contribue à de nombreux autres modules et projets *open-source*. Il maintient aussi un certain nombre de packages pour Mandriva (et Mageia maintenant).

Philippe Bruhat vit à Lyon et travaille pour Booking.com. Utilisateur de Perl depuis 1998, il est l'auteur de vingt-cinq modules sur CPAN et d'articles dans *GNU/Linux Magazine France* ; il est aussi cotraducteur de *Programming Perl* et *Perl Best Practices*, contributeur à *Perl Hacks* et relecteur pour O'Reilly. Il a participé à (et organisé) de nombreuses conférences Perl depuis 2000.

Avant-propos

Le langage Perl est en plein essor. Plus de vingt ans ont passé depuis sa création, et pourtant il a su s'adapter pour rester un langage pertinent.

Car depuis mi-2008, c'est une petite révolution de fond qui a lieu. S'inspirant des concepts de la version 6 à venir, elle même s'inspirant d'autres langages, Perl 5 s'est radicalement modernisé.

Tout d'abord, depuis la version 5.10, l'interpréteur `perl` est mis à jour régulièrement, avec une version majeure tous les ans apportant de nouvelles fonctionnalités et des mises à jour mineures tous les trimestres. Ce cycle régulier est garant d'un langage toujours stable et à jour des dernières nouveautés.

Ensuite, le langage lui-même a subi de profondes modifications, avec notamment un nouveau système de programmation orientée objet, `Moose`, et de nouveaux mots-clés.

Le dépôt de modules CPAN voit également son rythme de croissance s'accélérer, malgré sa taille déjà confortable (21 000 distributions et 86 000 modules). Aucun autre langage dynamique ne possède autant de bibliothèques externes centralisées.

Enfin, un nombre croissant de logiciels Perl de haut niveau et utilisant les dernières technologies voient le jour, dans tous les secteurs. Dans le domaine du Web notamment, Perl amorce son retour sur le devant de la scène, avec des *frameworks* complets comme Catalyst, ou plus légers et « agiles » comme Dancer.

Bien sûr, toutes ces évolutions changent la manière dont Perl est utilisé : un programme écrit en l'an 2000, même s'il reste compatible, ne ressemble plus du tout au même programme écrit au goût du jour, en 2010. C'est donc tout l'intérêt de ce livre, véritable *guide de survie* dans le monde Perl contemporain : les auteurs, fortement impliqués dans l'évolution de ce langage, vous apprennent à l'utiliser dans ses dimensions les plus actuelles.

Un monde qui vous étonnera par sa richesse et ses possibilités. . . mais surtout par son incomparable modernité !

1

Démarrer avec Perl

Cet ouvrage contient un nombre important d'exemples et d'extraits de code. La plupart constituent des petits programmes à part entière, qu'il est possible d'exécuter directement, d'autres sont des extraits non fonctionnels tels quels, mais qui sont suffisamment simples pour être intégrés dans des programmes existants.

Ce petit chapitre a pour but de démarrer rapidement avec cet ouvrage, pour pouvoir exécuter les exemples de code immédiatement et expérimenter le langage.

En règle générale, Perl est disponible sous Linux et Mac OS X. Sous Windows toutefois, il faut commencer par l'installer.

Pour des détails quant à l'installation de Perl, voir page 5.

Vérifier la version de Perl

Il est très important de s'assurer de disposer d'une version récente de Perl. En effet, cet ouvrage se focalise sur les versions modernes de Perl, c'est-à-dire dont la version est supérieure ou égale à 5.10.

```
perl -v
```

Cette commande renvoie quatre paragraphes de description et d'information sur Perl. La première ligne ressemble à cela :

```
This is perl, v5.10.0 built for darwin-2level
```

Elle indique qu'il s'agit de la version 5.10 de Perl. Si la version était inférieure à 5.10, il faudrait mettre à jour Perl (pour installer ou mettre à jour Perl, voir page 5).

Exécuter les exemples

Le moyen le plus simple de tester les concepts décrits dans cet ouvrage est d'exécuter les exemples de code. Pour cela, il est conseillé de les reproduire dans un fichier, puis d'utiliser l'interpréteur Perl en lui donnant le fichier en argument.

La première étape consiste à créer un fichier (qui sera nommé *test.pl*[1]). Ce fichier doit contenir les lignes suivantes, au début :

```
use strict;
use warnings;
use 5.010;
```

Ces lignes précisent à l'interpréteur Perl qu'il doit passer en mode strict et activer tous les avertissements, et qu'il requiert une version supérieure ou égale à 5.10, dont il active toutes les fonctionnalités supplémentaires.

En dessous de ces lignes, le code Perl à proprement parler peut enfin être inséré.

1. L'extension de fichier *.pl* est standard pour les programmes autonomes Perl, également appelés *scripts*. Les modules Perl, qui doivent être chargés, sont stockés dans des fichiers avec l'extension *.pm*.

Voici un exemple d'un tel fichier, pour exécuter l'extrait de code sur `foreach` (voir page 66) :

```
### fichier test.pl ###

use strict;
use warnings;
use 5.010;

# tout l'alphabet
my @alphabet = ('a'..'z');

my $count = 0;
foreach my $lettre (@alphabet) {
  say 'la ' . ++$count .
      'e lettre de l\'alphabet est : ' .
      $lettre;
}
```

Exécuter `perl` sur un fichier

Il ne reste plus qu'à lancer `perl` en lui passant le fichier *test.pl* en argument :

```
perl test.pl
```

Et la console affiche le résultat de l'exécution, dans ce cas précis :

```
la 1e lettre de l'alphabet est : a
la 2e lettre de l'alphabet est : b
la 3e lettre de l'alphabet est : c
...
la 25e lettre de l'alphabet est : y
la 26e lettre de l'alphabet est : z
```

Attention

Il est très important d'ajouter les trois lignes :

```
use strict;
use warnings;
use 5.010;
```

au début du fichier. Faute de quoi l'interpréteur ne prendra pas en compte les nouvelles fonctionnalités de Perl 5.10, notamment la fonction `say()`.

Éditer les programmes

Pour éditer les programmes Perl, il est nécessaire d'utiliser un éditeur de texte puissant et si possible orienté programmation, qui prenne en charge les spécificités de Perl.

L'environnement de développement intégré (IDE) Eclipse peut être utilisé avec le greffon (*plugin*) EPIC. Le site web **www.epic-ide.org** contient les instructions (très simples) qui permettent d'installer EPIC dans Eclipse. Bien sûr, le prérequis est d'avoir une version récente d'Eclipse, qui peut être téléchargée sur **www.eclipse.org**. Cette solution EPIC-Eclipse est multi-plateforme, et fonctionne très bien sous Linux, Windows ou Mac OS X.

Padre (**http://padre.perlide.org**) est un nouvel IDE écrit en Perl, par des développeurs Perl, pour les développeurs Perl. En très peu de temps, cet IDE a acquis une renommée certaine, et c'est probablement l'IDE Perl de demain. Il fonctionne sous Windows, Linux, et Mac OS X[2]. Pour Windows, il y a même un paquetage qui intègre Strawberry Perl et Padre en une seule installation, qui permet d'obtenir un environnement Perl opérationnel en quelques minutes et sans aucune configuration.

2. Même si l'installation sur ce système est moins aisée que sur les deux premiers.

Pour les utilisateurs qui n'utilisent pas d'IDE, voici une petite liste très classique de recommandations :

- Sous Unix, Emacs et vim sont incontournables. D'autres éditeurs de texte (comme SciTE) sont également couramment utilisés.
- Sous Windows, notepad++ et Editplus sont des éditeurs qui proposent une prise en charge basique de Perl. Vim est également disponible sous forme graphique, gvim.
- Sous Mac OS X, il y a une version Emacs de très bonne qualité, ainsi que des éditeurs spécifiques à cette plateforme, comme TextMate.

Installer Perl sous Linux

Pratiquement toutes les distributions Linux sont fournies avec Perl en standard. Toutes les distributions majeures (Ubuntu, Debian, Red Hat, Fedora, Mandriva, OpenSuSE...) permettent d'utiliser Perl directement sans avoir à installer quoi que ce soit.

Cependant il convient de vérifier la version installée, avec :

```
perl -v
```

qui permet de renvoyer des informations sur la version de l'interpréteur `perl`, dont la version doit être au moins 5.10. Si ce n'est pas le cas, il convient de mettre à jour Perl.

Pour mettre à jour Perl sous Linux, il suffit d'utiliser le gestionnaire de paquetage livré avec la distribution.

```
# Ubuntu / Debian
aptitude update && aptitude install perl5
# Mandriva
urpmi perl
# Red Hat / Fedora
yum install perl
```

Installer Perl sous Windows

Perl n'est pas livré en standard avec Windows. Il existe plusieurs distribution Perl pour Windows, mais la meilleure à ce jour est Strawberry Perl. En se connectant sur **http://strawberryperl.com**, il est aisé de télécharger la dernière version. Strawberry Perl s'installe très facilement[3], il possède un programme d'installation MSI qui guide l'utilisateur tout au long de la configuration.

Une fois installé, le répertoire contenant l'interpréteur `perl` est ajouté à la variable d'environnement `PATH`, et `perl.exe` est normalement utilisable depuis n'importe quel répertoire dans la console Windows. Il peut également être utilisé à travers des IDE, comme Padre ou Eclipse.

Installer Perl sous Mac OS X

Perl est fourni en standard avec Mac OS X. Cependant, il est préférable de mettre à jour Perl. Pour cela, la solution la plus simple est d'utiliser le projet MacPorts (**http://www.macports.org**), qui est un système de paquetage similaire à ceux disponibles sous Linux.

Il faut tout d'abord installer le logiciel MacPorts sur le Mac, puis mettre à jour Perl. MacPorts est une grande bibliothèque de logiciel, et contient plusieurs paquetages (*ports* dans la dénomination MacPorts) de Perl : `perl5`, `perl5.8`, `perl5.10`, `perl5.12`, et probablement d'autres plus à jour. Il est conseillé d'installer la dernière version, ou `perl5.12` au minimum. Voici la ligne de commande à entrer pour installer `perl5.12` :

```
port install perl5.12
```

L'interpréteur pourra être lancé de `/opt/local/bin/perl`.

3. Par défaut sur `C:/strawberry`, configurable pendant l'installation.

2

Installer un module Perl

CPAN (*Comprehensive Perl Archive Network*) est la plus vaste bibliothèque logicielle Perl. En pratique, c'est un ensemble de sites web qui donnent accès en ligne à la formidable bibliothèque des modules Perl – ainsi qu'à leur documentation –, modules qui permettent d'élargir les possibilités du langage Perl.

Attention

Il ne faut pas confondre :

- CPAN : le nom de la bibliothèque logicielle mise à disposition du public sur Internet ;
- `cpan` : le nom du programme à lancer en ligne de commande pour pouvoir chercher et installer des modules depuis CPAN.

De même, il ne faut pas confondre :

- Perl (avec une majuscule) : le langage de programmation ;
- `perl` (sans majuscule) : le nom de l'interpréteur `perl`, c'est-à-dire le programme qui exécute les programmes.

CPAN

CPAN est un élément capital du succès de Perl. Aucun autre langage ne possède une telle bibliothèque, vaste, bien organisée, bien gérée, compatible, et soutenue par une communauté aussi performante.

En particulier, le site Search CPAN (**http://search.cpan.org**) est la pierre angulaire pour trouver des modules, accéder à leur documentation en ligne, ainsi qu'à des commentaires et ajouts faits à la documentation, au code source des modules, aux archives des distributions, aux tickets ouverts, aux appréciations des utilisateurs, aux rapports des tests automatiques (très utiles pour savoir sur quelles versions de Perl et sur quelles plateformes le module fonctionne) et d'autres outils annexes encore.

Installer un module CPAN est une opération qui consiste à trouver le module recherché dans la bibliothèque CPAN, le télécharger, le configurer, trouver s'il dépend d'autres modules non installés, et enfin à installer le module et ses éventuelles dépendances.

Heureusement, tout ce processus est très largement automatisé grâce à `cpan`. Ce programme se lance simplement depuis la ligne de commande :

```
cpan
```

Il est également possible d'exécuter cette ligne :

```
perl -MCPAN -e shell
```

Attention

Sous Linux et Mac OS X, il sera sans doute indispensable d'avoir les droits d'administrateur pour pouvoir installer un module Perl.

Il convient donc de passer en mode *root* avant de lancer cpan, ou bien d'utiliser la commande sudo.

Sous Windows, avec Strawberry Perl, `cpan` est automatiquement configuré. Cependant, sous Linux et Mac OS X, au premier lancement de `cpan`, plusieurs questions sont posées à l'utilisateur. Une réponse par défaut est toujours proposée, de sorte que la configuration initiale est très simple.

Une fois lancée, la commande `cpan` donne accès à un environnement de commande très simple. Les commandes sont entrées sur une ligne, et validées avec la touche Entrée.

Il est facile d'avoir la liste des commandes et leurs options en tapant `help` :

```
cpan[1]> help
```

Voici cependant les commandes les plus utilisées.

Chercher un module avec `cpan`

```
i /requete/
```

La commande `cpan i` permet de rechercher le nom exact d'un module[1] Perl en partant d'un fragment de son nom ou d'un de ses fichiers. Par exemple, il est possible d'avoir une liste de tous les modules Perl qui mentionnent XML en tapant :

```
cpan[1]> i /XML/
```

1. Ou d'un auteur, d'un *bundle*, d'une distribution.

Installer un module avec `cpan`

```
install Un::Module
```

Une fois le nom du module trouvé, il suffit d'utiliser la commande `install` pour l'installer :

```
install Modern::Perl
```

`cpan` va alors télécharger et installer le module. S'il dépend d'autre modules, l'utilisateur devra confirmer l'installation de ces dépendances.

Il existe une méthode plus directe d'installer un module. Au lieu de lancer depuis la console :

```
cpan
```

puis d'entrer :

```
install Un::Module
```

Il est possible d'entrer directement depuis la console :

```
cpan Un::Module
```

Cela va directement installer le module en question.

Mettre à jour les modules

```
r upgrade
```

La commande `r` permet de lister tous les modules dont il existe une mise à jour.

`upgrade` permet de mettre à jour un module, plusieurs, ou tous. Sans argument, `upgrade` met à jour tous les modules. Si un argument est donné, il servira de filtre.

Installer un module avec cpanm

```
cpanm Un::Module
```

Une nouvelle méthode pour installer un module Perl est apparue avec l'arrivée du programme `cpanm`, dont le module d'installation s'appelle `App::cpanminus`. `cpanm` permet d'installer très rapidement un module Perl, sans aucune interaction. Pour l'installer, il suffit d'utiliser `cpan` :

```
cpan App::cpanminus
```

ou encore plus simplement, de télécharger directement le programme :

```
wget http://xrl.us/cpanm
chmod +x cpanm
```

La commande `cpanm` est alors disponible et peut être utilisée en lieu et place de `cpan`. Pour installer un module, il suffit de taper :

```
cpanm Un::Module
```

`cpanm` n'a pas besoin d'être configuré et n'affiche pas d'information tant que l'installation se passe bien. Les modules dépendants sont automatiquement installés, ce qui fait que l'utilisateur n'a pas à interagir avec `cpanm`.

3

Éléments du langage

L'interpréteur `perl` s'exécute traditionnellement soit depuis la console, soit à travers un IDE.

Exécuter `perl` en ligne de commande

Il est possible de donner des instructions directement à `perl` sur la ligne de commande :

```
perl -E 'say "hello world";'
```

Astuce

Sous Windows, la console par défaut n'est pas très puissante, et ne comprend pas les guillemets simples `'`. On peut écrire la ligne de cette façon :

```
perl -E "say 'hello world';"
```

Ou bien utiliser une console de meilleure qualité, comme Tera Term[1].

1. Disponible sur **http://ttssh2.sourceforge.jp**.

Cette manière d'exécuter du code Perl est très utile pour effectuer des petites tâches, ou bien pour tester quelques instructions :

```
perl -E 'my $variable = 40; $variable += 2; say $variable'
```

Cette commande affiche :

```
42
```

qui est bien le résultat de `40 + 2`.

Info

L'option `-E` permet d'exécuter du code Perl fournit en argument de la ligne de commande. Cette option est identique à `-e` sauf qu'elle active toutes les fonctions des dernières versions de Perl. Il est donc recommandé de toujours utiliser `-E`, et non `-e`.

L'interpréteur supporte beaucoup d'autres options. Elles sont documentées dans le manuel *perlrun*[2], mais voici les plus importantes d'entre elles :

- `-h` : affiche un résumé des options de l'interpréteur.
- `-w` : active l'affichage des messages d'alertes (*warnings*). Cette option est conseillée car un message de *warnings* qui apparaît est souvent synonyme de *bug* potentiel.
- `-d` : lance l'exécution en mode déboggage. Très utile pour trouver la source d'un problème dans un programme (voir page 15).

Exécuter `perl` sur un fichier

```
perl fichier.pl
```

2. Accessible avec la commande `man perlrun`

Il est plus pratique et pérenne de sauvegarder un programme Perl dans un fichier et d'exécuter l'interpréteur `perl` sur ce fichier :

```
perl fichier.pl
```

Info

L'extension de fichiers des *programmes* Perl est `.pl`. L'extension de fichiers des *modules* Perl (voir page 43) est `.pm`.

Créer un fichier exécutable

Sous Unix (Linux, Mac OS X), il est possible de créer un fichier qui s'exécutera directement avec `perl` en faisant commencer le fichier par :

```
#!/usr/bin/perl
my $var = 40;
print $var + 2;
```

Il est également nécessaire de rendre le fichier exécutable avec la commande `chmod` :

```
chmod u+x fichier.pl
```

Le fichier est maintenant exécutable :

```
./fichier.pl
```

Exécuter en mode déboggage

```
perl -d fichier.pl perl -d -E '..'
```

Il est possible de lancer l'interpréteur perl en *mode déboggage*. Dans ce mode, l'interpréteur peut exécuter les instructions une par une, et l'utilisateur a la possibilité de vérifier la valeur des variables, de changer leur contenu, et d'exécuter du code arbitraire.

Astuce

Les IDE (voir page 4) permettent de débogguer un programme Perl de manière visuelle, interactive, dans l'interface graphique, sans avoir recours au mode console expliqué ci-dessous.

Eclipse, avec le mode EPIC, et Padre permettent ainsi à l'utilisateur de placer visuellement des points d'arrêt (*breakpoint*), d'exécuter pas à pas les programmes, et d'afficher le contenu des variables.

Une fois `perl` lancé en mode déboggage, l'invite de commande affiche ceci :

```
$ perl -d -E 'my $var = 40;'
...
main::(-e:1):  my $var = 40;
 DB<1>
```

Voici une sélection des commandes qu'il est possible d'entrer pour interagir avec le mode déboggage :

- `h` : permet d'afficher un récapitulatif des commandes.
- `l` : permet d'afficher le code source à l'endroit de l'instruction en cours d'exécution.
- `-` : permet d'afficher les lignes précédant l'instruction en cours.
- `n` : permet d'exécuter l'instruction en cours, sans rentrer dans le détail. Ainsi, si l'instruction est un appel de fonction, celle-ci sera exécutée entièrement avant que le mode déboggage s'arrête.

- `s` : contrairement à `n`, `s` exécute l'instruction en cours, mais s'arrête à la prochaine sous-instruction.
- `r` : exécute toutes les instructions jusqu'à la prochaine instruction `return`.
- `c` : continue l'exécution du programme sans s'arrêter.
- `b` : place un point d'arrêt sur la ligne en cours. Il est possible de donner un numéro de ligne, un nom de fonction complet ou une condition en paramètre.
- `q` : quitte le mode déboggage.

Règles générales du langage

Un programme Perl est une suite d'instructions, séparées par des point-virgules[3].

Les instructions sont sensibles à la casse, donc `Fonction` n'est pas la même chose que `fonction`.

Les noms de variables commencent par un *sigil*, c'est-à-dire un caractère spécial qui permet de reconnaître son type. Les chaînes de caractères sont généralement entourées des caractères `"` ou `'` (voir page 20).

Le langage propose des fonctions de base[4]. Il est bien sur possible de créer des fonctions, pour modulariser le code.

En plus des fonctionnalités de base, il existe des centaines de *modules* additionnels[5] qui permettent d'enrichir le langage. Il est bien sûr possible d'écrire ses propres modules,

3. Il est possible d'omettre le point-virgule lors d'une fin de bloc, ou une fin de fichier.

4. La liste complète des fonctions de base est accessible dans la documentation *perlfunc*, accessible sous Unix avec `man perlfunc`.

5. Les modules additionnels sont disponibles sur CPAN (**http://cpan.org**), et installables à l'aide de la commande `cpan` (voir Chapitre 2).

pour rassembler les fonctionnalités semblables d'un logiciel et les diffuser.

Astuce

Il est possible de passer à la ligne avant la fin d'une instruction. Un retour chariot ne signifie pas une fin d'instruction. Seule un point-virgule, une fin de bloc ou une fin de fichier permet de signifier une fin d'instruction.

Cette fonctionnalité est très utile pour améliorer la lisibilité d'un code source :

```
my @tableau = ( 'sur', 'une', 'ligne');
my @tableau2 = ( 'sur',
                 'plusieurs',
                 'lignes'
               );
```

Les types de données

Perl propose de base trois types de données : les *scalaires*, les *tableaux* et les *tables de hachage*. Ces types de données sont expliqués en détail au Chapitre 5 (voir page 47), cependant en voici une introduction sommaire.

Les *scalaires* sont un type de données qui regroupe les nombres (entiers ou flottants), les chaînes de caractères, et les références (voir page 37). Voici quelques exemples de scalaires : `42`, `-3`, `0.01`, `2e3`[6], `"une chaîne de caractères"`. Les variables contenant un scalaire sont identifiées par le *sigil* `$`. Par exemple, `$variable`, `$chaine`.

Les *tableaux* sont des regroupements de scalaires. Ils sont à taille variable, et leurs indices commencent à 0. Il n'est pas nécessaire d'initialiser un tableau (voir page 47 pour plus

6. Correspond à $2*10^3$, donc 2 000.

de détails). Les variables contenant un tableau sont identifiées par le *sigil* `@`. Par exemple, `@tableau`, `@elements`.

Aussi appelées *tableaux associatifs*, ou simplement *hash*, les tables de hachage sont des regroupements d'associations clé-valeur. Les clés sont des chaînes de caractères, et les valeurs sont des scalaires (voir page 47 pour plus de détails). Les variables contenant une table de hachage sont identifiées par le *sigil* `%`. Par exemple, `%hash`.

Initialiser une variable scalaire

```
my $variable = ..
```

Déclarer une variable scalaire est très simple, il suffit d'utiliser l'opérateur d'affectation = :

```
my $nombre = 42;
my $test = "Ceci est une chaîne";
```

Le mot-clé `my` permet de spécifier que la variable est locale, et non globale.

Déclarer une variable scalaire

```
my $variable
```

Une variable peut également être déclarée sans être initialisée :

```
my $variable;
```

Dans ce cas, `$variable` prend une valeur spéciale, `undef`, qui signifie que la variable n'est pas initialisée.

Afficher un scalaire

```
print()
say()
```

Pour afficher un scalaire, il est possible d'utiliser `print`. Le code suivant affiche le nombre `5` sur la console :

```
my $variable = 5;
print($variable);
```

Cependant, pour faire de même mais avec un retour chariot à la fin de la ligne, `say` est plus pratique :

```
my $texte = "Bonjour";
say($texte);
```

`say` est une nouveauté de Perl 5.10, et se comporte comme `print`, sauf qu'un retour chariot est ajouté à la fin.

Créer une chaîne de caractères

```
"chaine", 'chaine', q(chaine), qq(chaine)
```

Les chaînes de caractères appartiennent à la famille des scalaires. Il y a plusieurs méthodes pour créer une chaîne de caractères :

- À l'aide des guillemets doubles `"` : les guillemets doubles permettent de créer une chaîne de caractères avec *interpolation*.

Info

Il est important de ne pas faire trop de différence entre un entier, un flottant et une chaîne de caractères. En Perl, ils sont tous regroupés sous la bannière des scalaires.

```
my $chaine = 'Ceci est une chaine';
# contient : Ceci est une chaine;

my $var = 42;
my $chaine = "Le resultat est $var";
# contient : Le resultat est 42;
```

Les caractères \n, \r etc. sont interprétés. Ainsi, un certain nombre de séquences d'échappement sont reconnues et interprétées (\n, \r, etc.). Le caractère d'échappement est l'anti-slash (\).

```
print "Retour\nChariot"
```

affiche :

```
Retour
Chariot
```

Pour insérer le caractère d'échappement ou des sigils, il suffit de les protéger :

```
print "\$var Retour\\nChariot";
# affiche $var Retour \nChariot
```

- À l'aide des guillemets simples ' : les guillemets simples permettent de créer une chaîne de caractères sans *interpolation*. Ainsi, un nom de variable ne sera pas remplacé par sa valeur ; \n, \r etc. ne seront pas interprétés.

```
my $chaine = 'Ceci est une chaine';
# contient : Ceci est une chaine;

my $var = 42;
my $chaine = 'Le resultat est $var \n';
# contient : Le resultat est $var \n;
```

- À l'aide de l'opérateur q() : l'opérateur q() permet de créer une chaîne de la même façon que les guillemets

simples ', mais la limite de la chaîne correspond à la parenthèse fermante *balancée* :

```
print q(chaine avec ', ", $var)
# affiche : chaine avec ', ", $var

print q(chaine (1, 3) )
# affiche : chaine (1, 3)
```

- À l'aide de l'opérateur `qq()` : de manière similaire, l'opérateur `qq()` permet de créer une chaîne avec interpolation.

Astuce

Il est possible d'utiliser `q()` et `qq()` avec d'autres caractères, par exemple `[]` ou `{}`. Cela permet de créer des chaînes qui contiennent des parenthèses non fermées et des guillemets :

```
print q[une d'erreur ("]
      . $erreur .
      q[")est apparue];
```

La notion de contexte

Une spécificité de Perl est la notion de contexte : Le comportement des opérateurs et l'évaluation des expressions dépendent du contexte dans lequel ils sont interprétés. Les contextes les plus souvent utilisés sont :

- **Le contexte scalaire numérique**. Une expression évaluée dans ce contexte est considérée comme un scalaire numérique, c'est-à-dire un nombre. Dans ce contexte, un opérateur aura un comportement adapté aux nombres. Par exemple, en contexte numérique, une chaîne `"03.20"` sera interprétée comme le nombre `3.2`. Attention, en contexte numérique, une liste ou un tableau sera interprété comme un nombre correspondant

au nombre d'éléments. En contexte scalaire numérique, la valeur `undef` renvoie 0.

- **Le contexte scalaire de chaîne**. Dans ce contexte, toute expression sera considérée comme une chaîne de caractères. Ainsi, un nombre sera interprété comme une suite de caractères. `3.2` sera interprété comme `"3.2"`. En contexte scalaire de chaîne, la valeur `undef` renvoie la chaîne vide `""`.

- **Le contexte de liste**. C'est le contexte qui permet de travailler avec des listes et des tableaux. En contexte de liste, un tableau renvoie sa liste d'élément. En contexte scalaire, un tableau renvoie un nombre (donc un scalaire) correspondant aux nombre d'éléments qu'il contient. Attention, en contexte de liste, `undef` renvoie une liste d'un élément contenant la valeur `undef`.

Info

Beaucoup d'opérateurs *forcent le contexte*. Une liste des opérateurs qui forcent le contexte scalaire numérique ou bien de chaîne est présentée ci-après (voir page 24).

D'autres opérateurs ne forcent pas de contexte. C'est le cas de l'opérateur d'affectation = qui fonctionne avec des scalaires numériques, des chaînes de caractères ou des listes, sans forcer de contexte. Dans ces cas-là, c'est le type des opérandes qui va influencer le contexte. Voici des exemples :

```
"5.00" + 3
```

Ici, l'opérateur + force le contexte numérique, et l'expression renvoie la valeur flottante 8.

```
my @tableau2 = @tableau1
```

Ici, la liste à gauche force l'évaluation de `@tableau1` en contexte de liste. `@tableau2` contient les éléments de `@tableau1`.

```
my $taille = @liste
```

Le scalaire à gauche force l'évaluation de `@liste` en contexte scalaire, ce qui renvoie la taille de la liste.

Il est possible de forcer un contexte de liste en utilisant des parenthèses `(..)`, et un contexte scalaire en utilisant `scalar()`.

```
my ($element1) = @liste;
```

Ici les parenthèses forcent le contexte de liste. `$element` contient donc le premier élément de `@liste`.

```
my @tableau2 = scalar(@tableau1);
```

`scalar` force l'évaluation de `@tableau1` en contexte scalaire, ce qui renvoie sa taille. Cette dernière est alors stockée en tant que premier élément dans `@tableau2`.

Travailler sur les nombres

```
+ - / * % ...
```

Perl propose les opérateurs mathématiques classiques : `+`, `-`, `*`, `/`, `sin`, `cos`, `%`.

`**` est disponible pour élever un nombre à une puissance.

Il existe des opérateurs raccourcis : `++`, `-`, `+=`, `-=`.

```
my $var = 5;
$var++;       #var vaut 6
$var--;       #var vaut 5
$var += 5;    #var vaut 10
$var -= 5;    #var vaut 5
$var *= 2;    #var vaut 10
```

Tous ces opérateurs forcent le contexte scalaire numérique.

Astuce

`$var++` renvoie la valeur de `$var`, puis incrémente `$var`, alors que `++$var` incrémente d'abord la variable, puis renvoie sa valeur. Ainsi, `return($var++)` est différent de `return(++$var)`.

La notion de précédence

Les opérateurs de Perl n'ont pas tous le même poids, ils sont plus ou moins « prioritaires ». On parle de leur *précédence*. Par exemple, l'opérateur multiplication `*` a une précédence plus forte que l'opérateur addition `+`.

Il est bien sûr possible de forcer l'application d'un opérateur avant un autre par l'usage de parenthèses :

```
my $resultat = 3 * (2 + 4);
# $resultat vaut bien 18
```

Il est utile de connaître la précédence des opérateurs pour éviter les parenthèses superflues, qui nuisent à la lisibilité (voir en annexe, page 429).

Travailler sur les chaînes

`. length chomp split join reverse substr index rindex`

L'opérateur `.` permet de concaténer deux chaînes de caractères. `chop` retire le dernier caractère d'une chaîne. `chomp` effectue la même opération mais seulement si le dernier caractère est un retour chariot.

`split` permet de séparer une chaîne par rapport à un motif (une expression régulière, voir page 122). `join` permet

l'opération inverse :

```
my $chaine = "bcdaofiazz";
say(join(" ¦ ", split(/a/, $chaine)));
# affiche bcd ¦ ofi ¦ zz
```

`substr` permet de remplacer une portion de chaîne de caractère par une autre. `index` et `rindex` permettent de chercher une portion de chaîne de caractères dans une autre en partant respectivement du début ou de la fin :

```
my $chaine = "Un lapin";
substr($chaine, 3, 0, "grand ");
# $chaine vaut "Un grand lapin"
substr($chaine, 0, 2, "Ce");
# $chaine vaut "Ce grand lapin"
my $var = index($chaine, "lapin");
# $var vaut 9
```

Tous ces opérateurs forcent le contexte de chaîne.

Tester si un scalaire est défini

```
defined
```

Pour tester si un scalaire contient une valeur, on doit utiliser `defined`, qui renvoie vrai si la variable est définie.

Déclarer une fonction

```
sub fonction { }
```

Une fonction se déclare grâce au mot-clé `sub`, suivi d'un nom et d'un bloc de code :

```
ma_fonction();
sub ma_fonction { say "Dans ma fonction" }
ma_fonction();
```

Cet extrait de code affiche deux fois le texte `Dans ma fonction`. Appeler une fonction se fait simplement en utilisant son nom, suivi de parenthèses pour passer des paramètres.

Une fonction peut être utilisée avant ou après sa déclaration.

Passer des paramètres

```
fonction($param)
```

Lors d'un appel de fonction, les paramètres sont passés entre parenthèses à la suite du nom de la fonction, séparés par des virgules :

```
ma_fonction(12, "une chaîne", -50);
```

C'est en fait une liste de scalaires qui est passée en paramètres. Pour récupérer les arguments de la fonction, voici le code qu'il faut insérer au début de la déclaration de la fonction :

```
sub ma_fonction {
  my ($param1, $param2, $param3) = @_;
  say "deuxième paramètre : $param2";
}
```

`@_` est une variable spéciale. C'est un tableau qui contient les paramètres reçus par la fonction. La première ligne de la fonction récupère les éléments du tableau `@_`, et stocke

les trois premiers dans les trois variables `$param1`, `$param2`, `$param3`.

Attention

Il est important de ne pas oublier les parenthèses autour de la liste des variables utilisées pour stocker les paramètres, et ce même s'il n'y a qu'un paramètre à récupérer :

```
my ($param1) = @_;
```

Renvoyer une valeur de retour

```
return $valeur
```

Une fonction Perl peut bien sûr renvoyer une valeur de retour, en utilisant `return` :

```
sub fois_dix {
  my ($param) = @_;
  return $param * 10;
}
my $resultat = fois_dix(2);
say $resultat;
```

Le mot-clé `return` termine l'exécution de la fonction en cours, et renvoie la ou les valeurs passés en paramètres. L'appelant peut récupérer la valeur de retour en l'assignant à une variable, ou bien en l'utilisant directement.

Astuce

Une fonction peut renvoyer plus d'une valeur, il suffit de renvoyer une liste :

```
my ($mois, $annee) = fonction();
sub fonction {
```

```
  return(12, 1957);
}
```

Dans cet exemple, la variable `$mois` aura pour valeur 12, et `$annee` 1957.

Utiliser des variables dans une fonction

```
my $variable
```

Lorsqu'une variable locale est déclarée dans une fonction, elle est invisible depuis l'extérieur :

```
my $annee = 1900;
say fonction(); # affiche 1800
say $annee;     # affiche 1900

sub fonction {
  my $annee = 1800;
  return $annee;
}
```

Dans la fonction, la portée de la variable `$annee` est limitée, et sa valeur n'est donc pas propagée au-delà de `fonction`.

Les opérateurs de test

```
== != eq ne
```

Il existe un nombre relativement grand d'opérateurs de test en Perl. Ces opérateurs sont divisés en deux catégories, ceux qui forcent un contexte scalaire numérique, et ceux qui forcent un contexte scalaire de chaîne.

Les opérateurs ==, !=, <, >, <=, >= permettent de tester respectivement l'égalité, la différence, l'infériorité, la supériorité, sur des valeurs numériques. Pour ce faire, ces opérateurs forcent le contexte scalaire numérique.

Vrai et faux en Perl

Toute expression est soit vraie ou fausse, selon sa valeur. En Perl, comme la valeur d'une expression peut dépendre du contexte en cours, voici la liste des cas où une expression est vraie ou fausse.

- *Nombres* : un nombre est vrai s'il est différent de zéro.
- *Chaînes de caractères* : une chaîne de caractères est *fausse* si elle est vide, c'est-à-dire qu'elle ne contient pas de caractère. Cependant, la chaîne de caractères `"0"` est également *fausse*. Dans les autres cas, une chaîne de caractères est une expression *vraie*.
- *Listes, tableaux* : une liste ou un tableau sont *vrais* s'ils contiennent au moins un élément. Une liste vide `()` ou un tableau sans élément sont *faux*.
- *Tables de hachage* : c'est une très mauvaise idée de tester si une table de hachage est *vrai* ou *faux*. Il faut toujours utiliser la fonction `keys` qui renvoie la liste des clés, ce qui permet de se replacer dans le cas d'une liste (voir page 80).
- `undef` : en contexte scalaire numérique, `undef` est évalué comme 0, donc *faux*. En contexte scalaire de chaîne, `undef` est évalué comme chaîne vide `""`, donc *faux*. En contexte de liste, `undef` est évalué comme une liste contenant un élément non définie, donc une liste de taille 1, donc *vrai*.

Il existe des opérateurs équivalents en contexte de chaînes, qui permettent de tester l'égalité, la différence, l'infério-

rité, la supériorité de deux chaînes entre elles : `eq`, `ne`, `lt`, `gt`, `le` et `ge`. Ces opérateurs forcent le contexte scalaire de chaîne.

La liste complète des opérateurs est disponible en annexe (voir page 429).

Tester une expression

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

Tester une expression se fait simplement avec l'opérateur `if`. Voici la forme traditionnelle, dite « forme préfixe ».

```
if ( $number > 5 ) {
  say "le nombre est plus grand que 5";
}
```

Il est possible d'ajouter un bloc `else`, qui est exécuté si le test est faux :

```
if ( $number > 5 ) {
  say "le nombre est plus grand que 5";
} else {
  say "le nombre est plus petit ou égal à 5"
      ;
}
```

Il est également pratique de chaîner les tests, avec `elsif` :

```
if ( $number > 5 ) {
  say "le nombre est plus grand que 5";
} elsif ($number < 5 ) {
  say "le nombre est plus petit que 5";
} else {
  say "le nombre est égal à 5";
}
```

Quelques remarques sur cette forme de l'opérateur `if` :

- Les parenthèses autour de l'expression à tester sont obligatoires.
- Les accolades sont également obligatoires autour des blocs de code, et ce même si le bloc ne contient qu'une ligne.

Bien évidemment, l'expression à tester peut être très complexe, comporter des appels de fonctions, etc. L'opérateur `if` interprète l'expression en *contexte scalaire*, comme l'illustre l'exemple suivant :

```
# une liste contenant un élément,
# l'entier zéro
my @list = (0);
if (@list) {
  say "la liste est non vide";
}
```

L'exemple de code précédent affiche bien à l'écran `la liste est non vide`. En effet, la liste `@list` contient un élément, l'entier 0. Cependant, en contexte scalaire, une liste renvoie le nombre d'éléments qu'elle contient. Donc `@list` en contexte scalaire renvoie `1`, qui est vrai, et donc le test est vrai.

Effectuer une opération conditionnelle

```
do_something() if $expression
```

Il existe une autre forme de l'opérateur `if`, appelée « forme infixe ». L'expression à tester est placée à la droite du mot-clé `if`, et le code à exécuter si le test est validé, à gauche du `if`. Cet ordre peut paraître incongru, mais correspond en fait à l'ordre logique linguistique.

En effet il est courant d'entendre en français : « Ouvre la fenêtre s'il fait trop chaud ». Ce qui peut s'écrire en Perl :

```
open_window() if too_hot();
```

Les intérêts de cette forme sont notamment :

- **Parenthèses optionnelles**. Les parenthèses autour de l'expression à tester ne sont pas obligatoires.
- **Pas d'accolades**. Le code à exécuter ne doit pas être entouré d'accolades ;
- **Lisibilité**. La structure du code ressemble plus à une grammaire linguistique.

Cependant, cette forme est probablement moins facile à appréhender par des personnes peu expérimentées. En sus, cette forme ne permet pas l'utilisation de `else`.

Effectuer une opération si un test est faux

```
do_something unless $expression
```

L'inverse du `if` infixe est l'opérateur `unless` infixe, qui peut être interprété comme « sauf » :

```
say "'$number' est positif" unless $number
    <= 0;
```

Un autre exemple utilisant l'opérateur modulo `%` :

```
say "le nombre est pair" unless $number % 2;
```

Ce programme affiche `le nombre est pair` sauf si `$number % 2` renvoie vrai, ce qui est le cas si `$number` est impair.

Cet exemple illustre le fait que l'utilisation de `unless` peut rendre la compréhension du code difficile.

`unless` est équivalent à `if !` :

```
say "le nombre est pair" if ! ($number % 2);
```

Tester négativement une expression

```
unless(. . .) {. . .}
```

L'opérateur `unless` existe également sous « forme préfixe ». Là aussi, il est équivalent à `if !` :

```
unless ($number % 2) {
  say "'$number' est pair";
  do_something();
}
```

Effectuer des tests avec `and` et `or`

```
and
or
```

En Perl, il est très important de connaître la précédence des opérateurs principaux, et les bonnes pratiques veulent que les parenthèses inutiles soient évitées. Il existe un couple d'opérateur intéressant : `and` et `or`.

`and` et `or` sont identiques à `&&` et `||`, mais ont une très faible précédence. Ils sont interprétés en bout de chaîne

de l'évaluation d'une expression, et de ce fait, sont utilisés comme charnières entre les morceaux de code à exécuter. Ainsi, les opérateurs `and` et `or` sont utilisés à la place de `if` et `unless`.

Voici un exemple d'utilisation de `or` :

```
do_something() or die "erreur : $!";
```

Ce code peut être décomposé en deux blocs, de part et d'autre de `or`.

Le bloc_1 : `do_something()`

Le bloc_2 : `die "erreur : $!"`

L'interpréteur de Perl va évaluer la ligne comme :

```
bloc_1 or bloc_2
```

Si bloc_1 renvoie vrai, alors l'ensemble de l'expression est vrai, donc bloc_2 ne sera pas exécuté. Si bloc_1 renvoie faux, alors il faut évaluer bloc_2. Donc, si l'appel de fonction do_something() renvoie faux, le programme s'arrête et affiche l'erreur.

De manière similaire, on peut voir ce type de code :

```
do_something() and say "success!";
```

Ce programme affichera `success!` si `do_something()` renvoie vrai.

Boucler sur les éléments d'une liste

```
foreach(. . .) {. . .}
```

Il existe deux types principaux de boucles. La boucle `foreach` permet d'effectuer des opérations pour chaque éléments d'une liste donnée, et la boucle `while` s'exécute tant qu'un test est vrai (voir section suivante).

```
foreach my $element (1, 2, "hello") {
  say $element;
}
```

Cet exemple de code utilisant `foreach` affiche successivement les éléments passés à la liste `foreach`. Il est possible d'omettre la variable temporaire (ici `$element`). Dans ce cas, l'élément en cours est stocké dans la variable spéciale `$_` :

```
foreach (1, 2, "hello") {
  say $_;
}
```

Le mot-clé `last` permet de sortir prématurément d'une boucle en cours. `next` quant à lui, oblige la boucle à passer à la prochaine itération immédiatement.

Boucler tant qu'un test est vrai

```
while(. . .) {. . .}
```

Les boucles `while` évalue la valeur de l'expression passée en paramètre ; tant qu'elle est vraie, le bloc de code est exécuté.

```
my $chaine = "animal";
while (my $car = chop $chaine) {
  say $car;
}
```

Cet extrait de code affiche toutes les lettres de la chaîne `"animal"`, mais en partant de la fin. En effet, `chop` enlève le dernier caractère de `$chaine`, et le renvoie. `$car` va donc contenir les caractères de la chaîne un à un, en partant de la fin. Tant que `$chaine` contient au moins un caractère, la valeur de l'expression `my $car = chop $chaine` est celle de `$car`, qui est une chaîne non vide, donc *vrai*.

Quand `$chaine` est vide, `chop` renvoie une chaîne vide, qui est également stockée dans `$car`. Dans ce cas, l'évaluation de l'expression `my $car = chop $chaine` est la chaîne vide, qui est *faux*. La boucle `while` s'arrête alors immédiatement.

Créer une référence sur scalaire

```
\$var
```

Voici comment créer une référence sur un scalaire :

```
my $variable = 5;
my $ref = \$variable;
```

La variable `$ref` est maintenant une référence sur `$variable`.

Info

En plus des nombres et des chaînes de caractères, il existe un autre type de scalaire : les *références*. Une référence est une variable qui *pointe* sur une autre variable. Cela permet notamment de passer une variable *par référence*.

Il est possible de créer une référence sur un scalaire, sur une liste, ou sur une table de hachage. Ces deux derniers types de réfé-

rences sont abordés dans le chapitre sur les structures de données (voir pages 67 et 84).

Déréférencer une référence sur scalaire

```
$$var
```

Déréférencer une référence se fait en ajoutant le *sigil* $ devant la variable référence :

```
my $variable = 5;
my $ref = \$variable;
my $variable2 = $$ref;
# $variable2 vaut 5
```

Accéder à une variable référencée

```
$$ref
```

Le double sigil $$ devant la référence permet d'accéder à la valeur de la variable référencée directement, pour récupérer sa valeur, ou la mettre à jour.

```
my $variable = 5;
my $ref = \$variable;
$$ref = 6;
say $variable;
# affiche 6
```

Passer un paramètre par référence

Lors d'un appel de fonction, il n'est généralement pas utile de passer des scalaires par référence. En effet, il suffit de les renvoyer par valeur de retour de la fonction :

```
my $var1 = 5;
my $var2 = 2;
my ($var1, $var2) = modif_valeurs($var1,
   ➥$var2);
say "résultat : $var1, $var2";

sub modif_valeurs {
  my ($var1, $var2) = @_;
  $var1++;
  $var2--;
  return ($var1, $var2);
}
```

Cet exemple affichera :

```
résultat : 6, 1
```

Cependant, il est également possible de passer un scalaire par référence :

```
my $var1 = 5;
my $var2 = 2;
modif_valeurs(\$var1, \$var2);
say "résultat : $var1, $var2";

sub modif_valeurs {
  my ($ref_var1, $ref_var2) = @_;
  $$ref_var1++;
  $$ref_var2--;
}
```

Cet extrait de code a exactement le même effet, sauf qu'il n'est pas nécessaire de renvoyer les deux variables en valeurs de retour de la fonction `modif_valeurs`. En effet,

leur références sont passées en paramètres, et la valeur des variables pointées par les références sont modifiées directement, avec `$$ref_var1++`.

Utiliser des références sur fonctions

Il est possible de créer des références sur fonctions anonymes. Ce mécanisme est très puissant, car il permet de passer des portions de code en paramètres, puis de les exécuter.

Créer une référence sur fonction se fait ainsi :

```
my $ref_transform = sub {
  my ($var) = @_; $var++; return $var;
};
```

Attention

Le caractère point-virgule ; est *obligatoire* à la fin de la définition de `sub`. La référence sur fonction définie est stockée dans la variable `$ref_transform`, c'est donc une allocation de variable, qui doit se terminer par un point-virgule.

À présent, la variable `$ref_transform` contient une référence sur une fonction anonyme. Il est possible de la manipuler de diverses manières. Elle peut par exemple être passée à une autre fonction comme un simple scalaire.

Pour exécuter la fonction référencée, il faut utiliser cette notation :

```
$ref_transform->(42);
```

L'opérateur *flèche* `->` permet de déréférencer et d'exécuter la fonction, en passant `42` en paramètre.

Voici un exemple d'utilisation :

```
my $ref_transform = sub {
  my ($var) = @_; $var++; return $var;
};

say appliq_transform(12, $ref_transform);

sub appliq_transform {
    my ($var, $ref_fonction) = @_;
    my $result = $ref_fonction->($var);
    return $result;
}
```

Cet exemple renvoie 13. L'extrait de code peut paraître inutile, cependant l'intérêt majeur est que la fonction `appliq_transform` ne sait rien de la fonction qu'elle reçoit par référence. Il est possible de changer le comportement de `appliq_transform` en changeant juste `$ref_transform`.

Ce mécanisme est un aspect de la programmation fonctionnelle. Perl permet de manipuler non seulement des données, mais aussi des morceaux de programmes.

Récupérer les arguments de la ligne de commande

`@ARGV`

Lorsqu'un script Perl est lancé depuis la ligne de commande, il est possible de lui passer des paramètres :

```
./script.pl argument1 argument2
```

Pour pouvoir récupérer ces paramètres à l'intérieur du script, il suffit d'utiliser la variable spéciale `@ARGV` :

```
my ($param1, $param2) = @ARGV;
```

`@ARGV` est un tableau dont les éléments sont les valeurs des paramètres de la ligne de commande.

Info

Plutôt que d'essayer d'interpréter les paramètres de lignes de commandes complexes tels que `-file-name "/tmp/file" -flag -port=5432`, il est plus judicieux d'utiliser l'un des nombreux modules d'interprétation des paramètres de ligne de commande, tel que `Getopt::Long`.

Exécuter des commandes système

`system(), `...``

La commande `system` permet d'exécuter une commande externe. Les *back-quotes* `` `...` `` permettent d'exécuter une commande et de récupérer sa sortie standard sous forme de chaîne de caractères dans une variable.

```
system("echo exemple");
my $contenu = `ls /tmp`;
```

Terminer abruptement un programme

`die`

La fonction `die` permet d'arrêter le programme instantanément, et permet d'afficher un message d'erreur – passé en paramètre de `die` – sur la console.

```
open my $f, "/tmp/fichier"
  or die "erreur d'ouverture";
```

Créer un module Perl

package

Les modules Perl sont des extensions du langage, qui enrichissent les fonctionnalités de Perl (voir le Chapitre 2, dédié à l'installation des modules).

Un module Perl est un fichier avec une extension *.pm*. Il contient l'instruction spéciale `package`, qui permet définir un espace de nommage (*namespace*) pour ce module[7].

```
package Mon::Module

sub test {
  say "chaîne de test";
}

1;
```

L'exemple précédent est à stocker dans un fichier *Mon/Module.pm*. En effet, il y a une correspondance entre le nom du module et son nom de fichier. Les doubles deux-points `::` correspondent à un répertoire. Ainsi, `Mon::Module` correspond au fichier *Mon/Module.pm*.

Un module – ou *package* – Perl doit se terminer par une expression *vraie*, qui signifie que le module s'est correctement chargé. Traditionnellement, cela se fait en ajoutant la ligne `1;` à la fin du fichier.

7. Par abus de langage, comme la plupart des modules ne contiennent souvent qu'un seul espace de noms, les termes *module* et *package* sont souvent utilisés de manière interchangeable.

Toute fonction définie dans un module peut être appelée depuis un script Perl (fichier en extension *.pl*) pour peu que le module soit chargé dans le script.

Utiliser un module Perl

```
use Mon::Module;
```

Dans un script Perl, pour utiliser un module Perl, il suffit d'utiliser `use`, qui va charger le module dans l'environnement du script.

```
use Mon::Module;

Mon::Module::test();
# affiche "chaine de test"
```

Concrètement, `use` transforme le nom du module en nom de fichier. Ici `Mon::Module` est transformé en *Mon/Module.pm*. Ce fichier est recherché dans une liste de répertoires[8]. Si le fichier est trouvé, il est chargé, sinon, une erreur survient.

Il est possible d'ajouter un répertoire à la liste des répertoire de recherche de modules grâce à `use lib`.

Voici un exemple de code qui illustre cette fonctionalité. Ici, le module `Mon::Module` est stocké dans le fichier */home/user/perl/Mon/Module.pm*. L'instruction `use lib "/home/user/perl"` l'ajoute à la liste des répertoires de recherche.

8. Cette liste de répertoire est accessible, elle est stokée dans la variable spéciale `@INC` : `perl -E 'say for @INC'` renvoie la liste des répertoires d'où peuvent être chargés les modules.

```
use lib "/home/user/perl";
use Mon::Module

Mon::Module::test();
# affiche "chaine de test"
```

Les fonctions du module peuvent être appelées en accolant leur noms au nom du module.

Les modules Perl offrent énormément de fonctionnalités, mais il est plus pertinent d'utiliser la programmation orientée objet, qui permet d'utiliser classes, méthodes et instances, au lieu de modules et fonctions (voir la partie II consacrée à la programmation orientée objet).

4

Structures de données

Nous allons aborder dans ce chapitre les structures de données dites « simples » : listes, tableaux et tableaux associatifs (tables de hachage). Ces types de données complètent les scalaires dans l'utilisation basique de Perl, et sont indispensables à la programmation plus avancée, ainsi qu'à l'utilisation des modules externes CPAN.

Nous présenterons également une sélection de modules permettant d'étendre les opérateurs de base du langage, `List::Util` et `List::MoreUtils`).

Créer une liste

```
(1, 2, 3)
```

Une liste est, pour simplifier, un ensemble fini de scalaires. Voici quelques exemples de listes :

```
(1, 2, 3);
(-25, 42, 3.141592);
('une hirondelle', 3, 'un lapin', -.5);
```

Une liste peut contenir des entiers, flottants, chaînes de caractères, et même d'autres types – telles par exemple les références, voir pages 67 et 84 –, du moment que ce sont des scalaires.

Les éléments d'une liste sont ordonnés : `(1, 2, 3)` est différent de `(1, 3, 2)`, bien que ces listes contiennent les mêmes éléments.

Créer une liste avec un intervalle

..

Tant qu'il y a peu d'éléments, une liste est facile à créer :

```
(1, 2, 3, 4, 5);
('a', 'b', 'c', 'd');
```

Cependant, il existe l'opérateur intervalle, constitué de deux points successifs `..`, qui permet de créer facilement une liste : en donnant ses bornes, l'opérateur crée la liste correspondante, en interpolant les éléments manquants. Pour les nombres entiers, cela donne :

```
(1..5);
# renvoit (1, 2, 3, 4, 5);
```

L'opérateur intervalle ne fonctionne que dans l'ordre croissant, donc l'exemple suivant renvoie une liste vide :

```
(5..1);
# renvoit une liste vide
```

Comment créer une liste de nombres décroissants ? Nous verrons comment faire cela lorsque sera abordée la fonction `reverse` (voir page 60).

L'opérateur intervalle fonctionne également sur les chaînes de caractères :

```
('a'..'z');
# toutes les lettres de l'alphabet

('aa'..'bb');
# est équivalent à :
('aa', 'ab', 'ac', 'ad', 'ae', 'af',
 'ag', 'ah', 'ai', 'aj', 'ak', 'al',
 'am', 'an', 'ao', 'ap', 'aq', 'ar',
 'as', 'at', 'au', 'av', 'aw', 'ax',
 'ay', 'az', 'ba', 'bb');
```

Créer une liste de mots

qw()

Il est souvent très utile de créer des listes de mots, comme par exemple :

```
('avancer', 'reculer', 'tourner',
   ➥'démarrer');
```

Les listes de mots sont tellement courantes qu'il existe un opérateur dédié, qui facilite leur écriture : l'opérateur `qw`. Son nom vient de l'abréviation de « Quoted Word », et il permet d'éviter d'écrire les guillemets autour des mots, et les virgules entre eux :

```
qw(avancer reculer tourner démarrer);
```

On pourra donc écrire :

```
foreach my $action (
  qw(avancer reculer tourner démarrer)
) {
  # ...
}
```

Astuce

L'opérateur qw s'utilise généralement avec des parenthèses, mais il est possible d'utiliser d'autres délimiteurs. Les lignes suivantes sont équivalentes :

```
qw(avancer reculer tourner démarrer)
qw[avancer reculer tourner démarrer]
qw/avancer reculer tourner démarrer/
qw!avancer reculer tourner démarrer!
```

L'intérêt est évident lorsqu'il faut créer des listes de caractères non alphanumériques :

```
qw[ . ( ) ]
# renvoie ( '.', '(', ')' )
qw/[ ] ( ) /
# renvoie ( '[', ']', '(', ')' )
```

Créer un tableau à une dimension

```
@array = . . .
```

Pour faire simple, un tableau est une variable qui contient une liste. Le *sigil* d'un tableau est @ :

```
my @array;
```

Pour initialiser un tableau, une liste peut lui être assignée :

```
my @array  = ('fraise', 12, 'framboise');
my @array2 = (1..10);
my @array3 = qw( pomme fraise
                 framboise poire );
```

Si un tableau n'est pas initialisé, il contient par défaut la liste vide `()`.

```
my @array;
# équivalent à :
my @array = ();
```

Accéder aux éléments d'un tableau

`$array[n]`

Accéder à un élément d'un tableau se fait en utilisant l'indice de l'élément, entre crochets :

```
my @array = (121, 122, 123);
my $element = $array[0];
# $element contient 121;
$element = $array[1];
# $element contient 122;
$element = $array[ 1 + 1 ];
# $element contient 123;
```

Attention

Les indices des éléments commencent à 0. `$array[0]` est le premier élément, `$array[1]` est le deuxième élément, etc.

À la lecture de cet exemple, il est possible de formuler plusieurs remarques :

- Pour accéder à un élément d'un tableau, c'est le *sigil* `$`[1] qui est utilisé. Ainsi il faut écrire `$array[1]` pour accéder au deuxième élément, et non pas `@array[1]`.
- Un indice n'est pas obligatoirement un nombre, il peut être une expression, qui sera évaluée en *contexte scalaire* en tant qu'entier, et le résultat sera utilisé comme indice.

En application directe, voici comment afficher un élément d'un tableau :

```
say "le 3e élément est : $array[2]";
# équivalent à
say 'le 3e élément est : ' . $array[2];
```

Affecter un élément d'un tableau

```
$array[n] = . . .
```

Pour changer un élément, il suffit d'assigner une valeur à l'élément d'un tableau. Il est bien sûr possible de faire des calculs directement avec des éléments de tableaux :

```
my @array = (10, 20, 30);
$array[1] = 15;
# le tableau vaut à présent (10, 15, 30);
$array[2] = $array[0] * 2.5;
# le tableau vaut à présent (10, 15, 25);
```

1. Un bon moyen mnémotechnique est de considérer ce qui est obtenu lorsque l'expression est évaluée. Ici, c'est l'élément du tableau qui est désiré, donc un scalaire. Par conséquent, l'expression doit commencer par le *sigil* `$`.

```
$array[0] -= 5;
# le tableau vaut maintenant (5, 15, 25);
$array[ $array[0] ] = 12;
# le tableau vaut maintenant
# (5, 15, 25, undef, undef, 12);
```

Obtenir le premier élément d'un tableau

```
$array[0]
```

Comme la numérotation des indices commence à zéro, `array[0]` correspond au premier élément du tableau.

Obtenir le dernier élément d'un tableau

```
$array[-1]
```

Perl permet d'utiliser des entiers négatifs comme indices, qui permettent de parcourir les éléments d'un tableau en partant de la fin. Ainsi l'indice `-1` correspond au dernier élément d'un tableau, et ainsi de suite :

```
my @array = (10, 20, 30, 40);
say $array[-1];
# Affiche 40
say $array[-2];
# Affiche 30
$array[-2] -= $array[-3];
# le tableau vaut (10, 20, 10, 40);
```

Obtenir la taille d'un tableau

```
$size = @array
```

La taille d'un tableau est le nombre d'éléments qu'il contient. En Perl, il n'y a pas de fonction particulière pour obtenir la taille d'un tableau, il suffit de l'*évaluer en contexte scalaire*. Sous cette description en apparence compliquée, se cache une opération très simple. Soit le code suivant :

```
my $size = @array;
```

Attention

Une erreur classique est d'essayer d'utiliser la fonction `length` pour calculer la taille d'un tableau : *cela ne fonctionne pas*. En effet, `length` calcule la longueur d'une chaîne de caractères uniquement. `length` attend donc un scalaire en paramètre. Si on donne un tableau en argument, `length` va renvoyer la longueur de la représentation textuelle du dernier élément du tableau.

À gauche, un scalaire, `$size` ; à droite, un tableau. Le fait que l'expression de gauche soit un scalaire oblige l'expression de droite à être évaluée *en contexte scalaire*. Le tableau de droite, en contexte scalaire, renvoie le nombre d'éléments qu'il contient. C'est en effet le seul résultat utile qui peut être stocké dans un unique scalaire.

Astuce

On peut forcer le contexte scalaire grâce à la fonction `scalar` :

```
my $size = scalar(@array);
```

En cas de doute sur le contexte dans lequel est évalué le tableau ou la liste, il faut utiliser `scalar()` pour en obtenir la taille.

Assigner un élément en dehors du tableau

Assigner un élément en dehors des limites d'un tableau est parfaitement valide. Le tableau va être étendu d'autant d'éléments que nécessaire pour avoir la bonne taille. Les éléments créés en plus seront indéfinis.

```
my @array = (10, 'poire', 30, 40);
$array[4] = 50;
# @array vaut maintenant
# (10, 'poire', 30, 40, 50)

$array[7] = 80;
# @array vaut maintenant
# (10, 'poire', 30, 40, 50,
#  undef, undef, '80')
```

Tester les éléments d'un tableau

exists(..) defined(..)

Chaque élément d'un tableau étant un scalaire, tous les tests sur les scalaires s'appliquent aux éléments d'un tableau :

```
my @array = (10, 'poire', 30, 40);
if ($array[1] eq 'abricot') {
  # ...
}
if ($array[0] > 5) {
  # ...
}
```

Mais il est aussi possible de tester la présence et la définition d'un élément. La fonction `exists` renvoie vrai si l'élément existe ; `defined` teste si un élément est défini. L'exemple suivant explicite la nuance entre les deux concepts :

```
my @array = (10, 20, 30);
$array[4] = 50;
# @array vaut maintenant
# (10, 20, 30, undef, 50)
# ce test est vrai
if (exists $array[4]) {
  # ...
}

# ce test est faux
if (defined $array[4]) {
  # ...
}
```

Le test utilisant `exists` renvoie vrai car il y a un élément à l'indice 4. Cependant, cet élément n'est autre que `undef`, et donc le second test échoue.

Manipuler la fin d'un tableau

push(..) pop()

Il est courant de vouloir ajouter des éléments à un tableau : par exemple, calculer des résultats, et les ajouter à un tableau, puis renvoyer le tableau, comme liste de résultats. Bien sûr, il est possible d'ajouter un élément à la fin d'un tableau en déduisant le dernier indice de sa taille :

```
my @array = (10, 'fraise', 30);
my $size = @array; # renvoie 3
```

```
$array[$size] = 40;
# @array vaut (10, 'fraise', 30, 40)
```

Mais c'est un peu rébarbatif. Il est beaucoup plus rapide d'utiliser push, qui permet d'ajouter un élément à un tableau :

```
my @array = (10, 'fraise', 30);
push @array, 40;
# @array vaut (10, 'fraise', 30, 40)
```

Astuce

En fait, push permet d'ajouter plus d'un élément à un tableau. Ainsi, push est capable d'ajouter une liste à la fin d'un tableau.

```
my @array = (10, 'fraise', 30);
push @array, 40, 50;
# @array vaut (10, 'fraise', 30, 40, 50)
```

L'inverse de push est pop, qui permet d'enlever un élément de la fin d'un tableau. pop ne permet pas d'enlever plus d'un élément à la fois. L'élément qui est retiré du tableau est renvoyé. Donc, la valeur de retour de pop correspond au dernier élément du tableau :

```
my @array = (10, 'fraise', 30);
my $last_element = pop @array;
# @array vaut maintenant (10, 'fraise')
# et $last_element vaut 30
```

Manipuler le début d'un tableau

```
shift(..)
unshift(..)
```

push et pop sont utilisés pour manipuler la fin d'un tableau, il existe l'équivalent pour le début : shift retire le premier élément d'un tableau et le renvoie, et unshift permet d'ajouter une liste d'éléments au début du tableau.

```
my @array = (10, 'fraise', 30);
my $first_element = shift @array;
# @array vaut maintenant ('fraise', 30)
# et $first_element vaut 10

unshift @array, $first_element, 42;
# @array vaut (10, 42, 'fraise', 30)
```

Info

Avec unshift, les éléments sont ajoutés tous ensemble et non un par un, leur ordre originel est donc respecté.

Manipuler le milieu d'un tableau

```
splice(..)
```

Voici comment manipuler une zone arbitraire contigüe d'un tableau.

La fonction splice permet de sélectionner des éléments d'un tableau, pour les supprimer ou les remplacer par d'autres. splice prend en argument le tableau, un indice à partir duquel commencer la sélection, un entier représentant la taille de la sélection, et une liste d'éléments qui remplaceront la sélection le cas échéant. L'exemple suivant remplace les deuxième et troisième éléments par un seul élément :

```
my @array = (10, 'fraise', 30, 40);
splice @array, 1, 2, 'pomme';
# @array vaut maintenant (10, 'pomme', 40)
```

Il est également possible que la liste de remplacement soit plus grande que la sélection :

```
my @array = (10, 'fraise', 30, 40);
splice @array, 1, 2, 'pomme', 'poire',
   ➥'abricot';
# @array vaut maintenant
# (10, 'pomme', 'poire', 'abricot', 40)
```

Si la liste de remplacement est omise, `splice` va ôter la sélection du tableau uniquement.

```
my @array = (10, 'fraise', 30, 40);
splice @array, 1, 2;
# @array vaut maintenant (10, 40)
```

Si la longueur de la sélection est omise, `splice` va tronquer le tableau à partir de l'indice de départ.

```
my @array = (10, 'fraise', 30, 40);
splice @array, 2;
# @array vaut maintenant (10, 'fraise')
```

Astuce

L'indice de départ peut être négatif, pour sélectionner depuis la fin du tableau.

La longueur de sélection peut également être négative : `splice( @array, 3, -2)` enlèvera tous les éléments à partir du 4e, sauf les deux derniers du tableau.

Supprimer un élément d'un tableau

```
splice(..)
```

Il ne faut pas confondre « réinitialiser un élément » et « supprimer un élément ». Réinitialiser un élément, c'est le mettre à `undef`, ce qui s'effectue très simplement comme suit :

```
my @array = (10, 'fraise', 30);
$array[1] = undef;
# @array vaut maintenant (10, undef, 30)
```

Pour supprimer un élément d'un tableau, il faut enlever l'élément en question du tableau. Pour cela il est possible d'utiliser `splice` :

```
my @array = (10, 'fraise', 30);
splice @array, 1, 1;
# @array vaut maintenant (10, 30);
```

Inverser une liste ou un tableau

```
reverse(..)
```

Les listes et tableaux sont ordonnés. Il est possible de les inverser, c'est-à-dire de les retourner, de manière à ce que les éléments soient dans l'ordre opposé, en utilisant `reverse` :

```
my @array = (10, 'fraise', 30, 40);
@array = reverse @array;
# @array vaut (40, 30, 'fraise', 10);
```

Il est possible de faire la même chose sur les listes directement :

```
my @array = reverse (10, 'fraise', 30, 40);
# @array vaut (40, 30, 'fraise', 10);
```

Astuce

Précédemment, il a été dit qu'il était impossible de définir un intervalle décroissant en utilisant `..`. Cependant, grâce à `reverse`, c'est possible et facile :

```
('a'..'z');
# renvoie toutes les lettres
# dans l'ordre alphabétique

reverse ('a'..'z')
# renvoie toutes les lettres de l'alphabet
# de z vers a

reverse (1..3)
# est équivalent à
(3, 2, 1)
```

Aplatir listes et tableaux

L'aplatissement des listes est un élément important du langage Perl et il est primordial de bien comprendre ce mécanisme.

Examinons les listes suivantes :

```
( 1, 2, 'a', 'b' )
( 1, 2, ('a', 'b') )
( 1, (2, 'a'), 'b' )
( ( 1, 2, 'a'), 'b' )
```

Contre toute attente, ces listes sont équivalentes entre elles, et sont « aplaties ».

Ce principe est valable également pour les tableaux :

```
my @array1 = (10, 'fraise', 30, 40);
my @array2 = ('a', 'b', 'c');
my @array3 = (@array1, @array2);
# @array3 vaut (10, 'fraise', 30, 40, 'a',
   ➥'b', 'c');

@array3 = (@array3, 1, 2);
# @array3 vaut maintenant
# (10, 'fraise', 30, 40, 'a', 'b', 'c',
   ➥1, 2);
```

Il est donc impossible de créer des « tableaux de tableaux » en essayant de regrouper une liste de tableaux ensemble. Pour créer des tableaux à *n* dimensions voir page 67.

Manipuler une tranche de tableau

```
@array[a..b]
```

En plus de `splice`, Perl permet de travailler avec des *tranches* de tableau (*slice* en anglais). Une tranche de tableau est une sélection d'indices d'un tableau. Une tranche permet de récupérer ou d'assigner une partie d'un tableau. Il est intéressant de noter qu'une tranche n'est pas nécessairement contiguë. Les exemples suivants permettent de se familiariser avec ces *slices* :

```
# récupérer une partie contiguë d'un tableau
my @array = (10, 'fraise', 30, 40, 'pomme');
my @tranche = @array[ 1, 2 ];
# @tranche vaut ('fraise', 30)
```

```
@tranche = @array[ 1..3 ];
# @tranche vaut ('fraise', 30, '40');
```

Attention

Lors d'une utilisation d'une tranche de tableau, le sigil du tableau reste `@`.

Il est possible d'utiliser une tranche contigüe pour remplacer des éléments :

```
# remplacer des éléments contigus
my @array = (10, 'fraise', 30, 40, 'pomme');
@array[ 1, 2 ] = ('poire', 'citron');
# @array vaut maintenant
# (10, 'poire', 'citron', 40, 'pomme')
```

Attention

Contrairement à `splice`, il n'est pas possible de changer la taille d'une tranche. Les éléments en trop seront ignorés, et ceux manquants seront remplacés par `undef`.

```
my @array = (10, 'fraise', 30, 40, 'pomme');
@array[1, 2] = ('poire');
# @array vaut maintenant
# (10, 'poire', undef, 40, 'pomme')

@array[1, 2] = ('poire', 'citron', 'pêche');
# @array vaut maintenant
# (10, 'poire', 'citron', 40, 'pomme')
```

Cette propriété peut être utilisée pour réinitialiser (mettre à `undef`) une partie de tableau :

```
my @array = (10, 'fraise', 30, 40, 'pomme');
@array[ 1, 2 ] = ();
# @array vaut maintenant
# (10, undef, undef, 40, 'pomme')
```

En plus des listes, les tableaux peuvent être utilisés pour spécifier une tranche :

```
my @indices = (2, 3);
my @array = (10, 'fraise', 30, 40, 'pomme');
@array[ @indices ] = ('abricot', 'citron');
# @array vaut maintenant :
# (10,'fraise','abricot','citron','pomme')
```

Là où les tranches deviennent très puissantes, c'est qu'elles peuvent être discontinues. Voici un exemple qui travaille avec les nombres pairs et impairs :

```
my @pairs = (0, 2, 4, 6);
my @impairs = (1, 3, 5, 7);
my @array = (0..7);
@array[ @pairs ] = ( reverse(4..1) );
@array[ @impairs ] = ( reverse(8..5) );
# @array vaut maintenant :
# (4, 8, 3, 7, 2, 6, 1, 5)
```

L'opérateur virgule

Contrairement à ce qu'il paraît, et contrairement à d'autres langages, ce ne sont pas les parenthèses qui caractérisent une liste, mais c'est la virgule.

```
1, 2
```

est une liste valide. La virgule , est l'opérateur qui crée la liste[2]. Cependant, une liste est pratiquement toujours écrite avec des parenthèses autour des éléments, car l'opérateur virgule a une très faible précédence (voir le tableau de précédence des opérateurs, page 429).

2. En fait, l'opérateur virgule change de comportement suivant le contexte. En contexte de liste, la virgule crée une liste. En contexte scalaire, le dernier élément est renvoyé.

```
a = 12, 13;
# est équivalent à
(a = 12), 13;
```

Le tableau `@a` contiendra un unique élément (12). Pour connaître l'ordre de précédence, voir la liste des opérateurs page 429. On y voit que l'opérateur virgule est presque tout en bas de la liste, alors que l'opérateur = est au-dessus.

En conclusion : toujours utiliser des parenthèses autour des listes, mais savoir que c'est la virgule qui crée la liste.

Boucler sur les éléments d'un tableau

`foreach (. . .) {. . .}`

`foreach` permet d'effectuer des opérations en boucle. Il existe plusieurs syntaxes possibles, mais la syntaxe moderne est la suivante :

```
foreach my $variable ( expression ) {
  # corps de la boucle utilisant $variable
}
```

L'*expression* est interprétée en *contexte de liste*, et pour chaque élément de l'expression, le corps de la boucle est interprété, `$variable` valant l'élément en question[3]. L'exemple

3. En fait, `$variable.` n'a pas juste la valeur de l'élément en cours, c'est un *alias* dessus. Changer la valeur de `$variable` change l'élément, du moins si l'élément peut être modifié. Ainsi
`foreach my $v (1, 2) { $v++ }` renverra une erreur, mais
`foreach my $v (@j = (1,2)) { $v++ } say "@j"`
affichera `2 3`. Cependant, utiliser ce mécanisme n'est pas recommandé, car l'affectation, trop peu visible, est source de nombreux bogues.

naïf suivant affiche une liste au format HTML[4] :

```
say '<ul>';
# foreach avec une liste
foreach my $fruit ('pomme', 'fraise',
   ➥'framboise') {
  say "  <li>$fruit</li>";
}
say '</ul>';
```

Le code précédent utilise une simple liste de chaînes de caractères, qui sont affichés, encadrés par `<li>` et `</li>`. `foreach` fonctionne aussi sur les tableaux :

```
# tout l'alphabet
my @alphabet = ('a'..'z');

my $count = 0;
foreach my $lettre (@alphabet) {
  say 'la ' . ++$count .
      'e lettre de l\'alphabet est : ' .
      $lettre;
}
```

Bien sûr, il est possible d'évaluer n'importe quelle expression, et le résultat sera utilisé pour boucler :

```
my @entiers = (0..100);
my @alphabet = ('a'..'z');

foreach my $e (@entiers[7..11],
   ➥@alphabet[7..11]) {
  print "$e-";
}
# affiche 7-8-9-10-11-h-i-j-k-l-
```

4. Ce n'est pas une méthode recommandée pour générer du HTML de manière propre. Pour cela, utilisez par exemple le module `Template::Toolkit`.

Dans le code précédent, il faut se souvenir du concept d'aplatissement de liste, qui fait que `@entiers[7..11], @alphabet[7..11]` est vu comme une seule liste des entiers de `7` à `11`, et des lettres de `h` à `l` (voir page 61).

Créer un tableau à plusieurs dimensions

Les éléments d'une liste ou d'un tableau sont des scalaires. Il n'est donc pas possible de stocker un tableau (car ce n'est pas un scalaire) en tant qu'élément : le concept d'aplatissement s'applique (voir page 61).

Cependant, il y a un moyen très simple (et de nombreux outils syntaxiques) pour construire des tableaux à *n* dimensions, et d'autres structures plus compliquées, en utilisant des *références*.

Référencer un tableau

```
[1, 2, 3]
\@array
```

Une référence est un scalaire spécial, qui « pointe » sur autre chose, que ce soit un autre scalaire, un tableau, une table de hachage.

Pour obtenir une référence, il est possible d'utiliser \ sur un tableau, qui renvoie une référence pointant dessus. `[ ]` peut également être utilisé, cela permet de créer une référence sur une liste directement, ou sur un tableau déjà existant. Ces méthodes basiques sont résumées dans le code suivant :

```
my @array = (1, 2, 3);
my $ref_array = \@array;
# $ref_array est une référence sur @array

my $ref_array = [1, 2, 3];
# $ref_array est une référence sur (1, 2, 3)
```

Dans l'exemple précédent, `[1, 2, 3]` est appelé une *référence vers un tableau anonyme*.

Déréférencer un tableau

```
@{$ref_array}
```

Pour accéder à un tableau référencé, il faut utiliser `@{}` :

```
my $ref_array = [1, 2, 3];
my @array = @{$ref_array};
```

L'opérateur `@{}` évalue l'expression donnée, et déréférence le résultat en tableau. Lorsqu'il n'y a pas d'ambiguïté, les accolades peuvent être omises :

```
my $ref_array = [1, 2, 3];
my @array = @$ref_array;
```

Il est possible d'accéder directement à un élément d'un tableau en partant de sa référence, grâce à l'opérateur flèche `->` :

```
my $ref_array = [1, 2, 3];
my $value = $ref_array->[1];
# $value vaut 2
```

Créer des références dans un tableau

Les références sont très utiles pour créer des tableaux à deux dimensions ou plus. Au lieu d'essayer de stocker des tableaux dans des tableaux, il suffit de stocker des références dans un tableau simple.

Pour illustrer l'explication, voici un exemple, dans lequel il s'agit de créer un tableau à deux dimensions. Soit 3 étudiants, pour lesquels il faut stocker leur moyenne générale pour les 3 premiers mois de l'année. La première dimension du tableau sera donc les 3 premiers mois de l'année, de `0` à `2`. Dans la seconde dimension, donc pour chaque mois, seront stockées les moyennes générales des élèves.

Voici les données disponibles : le premier élève a eu 10 en janvier, 13 en février et 12.5 en mars ; le second élève a obtenu 12 le premier mois de l'année, puis 10, et enfin 7 en mars ; la moyenne du dernier élève était de 15 en janvier, 17 en février et 18 en mars.

Voici un moyen de stocker ces valeurs dans un tableau à deux dimensions :

```
# création des références sur tableaux
my $ref_month_1 = [10, 12, 15];
my $ref_month_2 = [13, 10, 17];
my $ref_month_3 = [12.5, 7, 18];

# ensuite, stockage dans un tableau général
my @students_averages = ( $ref_month_1,
                          $ref_month_2,
                          $ref_month_3
                        );
```

Cette manière de faire a l'avantage d'être simple et explicite, mais elle n'est pas très puissante. Nous aurions pu écrire directement le tableau final comme suit :

```
# définition et initialisation
# du tableau final directement
my @students_averages = ( [10, 12, 15],
                          [13, 10, 17],
                          [12.5, 7, 18],
                        );
```

Accéder à un tableau de tableaux

Pour cette section, le tableau `@students_averages` précédemment créé va être réutilisé. Un tableau à *n* dimensions est un tableau normal, dans lequel des références sont stockées. Logiquement, il est possible d'accéder à un de ses éléments en utilisant son indice. La valeur ainsi récupérée sera une référence, qu'il faudra déréférencer pour pouvoir l'exploiter.

Par exemple, pour récupérer la moyenne du deuxième élève au mois de mars :

```
my $ref_march = $students_averages[2];
my @march = @$ref_march;
my $average_in_march = @march[1];
```

Cependant, Perl permet d'utiliser le raccourci ->. Cet opérateur permet de déréférencer, puis d'accéder à un élément directement :

```
my $average_in_march =
   ➥$students_averages[2]->[1];
```

Une autre syntaxe est possible, en faisant disparaître purement et simplement la flèche :

```
my $average_in_march =
   ➥$students_averages[2][1];
```

Info

Certains développeurs préfèrent garder l'opérateur -> pour indiquer plus clairement qu'il s'agit d'un déréférencement, d'autres préfèrent la notation plus courte qui est similaire à d'autres langages.

Modifier un tableau de tableaux

Comme dans le cas d'un tableau plat, il suffit d'affecter une valeur pour modifier le tableau :

```
# le 3e élève a eu 11 en janvier, et non 15
$students_averages[0]->[2] = 11;

# ajout du mois d'avril
$students_averages[3] = [14, 5, 16];

# idem, mais cette fois en déréférençant
# à gauche
@{$students_averages[3]} = (14, 5, 16);
```

Dumper un tableau

« Dumper » est un anglicisme, qui vient du verbe *dump*, utilisé en anglais pour *« définir une action de vidage mémoire vers un périphérique de sortie, généralement à des fins d'analyse, effectuée suite à une exception ou erreur*[5] ». En pratique, « dumper » une structure permet de récupérer une chaîne de caractères qui la décrit, pour pouvoir l'afficher.

Le moyen le plus simple est d'utiliser un module standard fourni avec Perl, `Data::Dumper`. Il prend en argument la

5. **http://fr.wikipedia.org/wiki/Dump**.

structure à afficher, et renvoie une chaîne de caractères. Il est préférable de lui donner la référence sur la structure.

```
# Charge le module
use Data::Dumper;
# offre un affichage plus concis
$Data::Dumper::Indent = 0;

my @array = (1, 2, 3);
say Dumper(\@array);

# affiche à l'écran :
# $VAR1 = [1,2,3];
```

Info

`$Data::Dumper::Indent = 0` met la variable `$Indent` du module `Data::Dumper` à 0. Cette variable gère le niveau d'indentation de sortie de la méthode `Dumper`. Lorsque cette variable est initialisée à zéro, le texte produit est très concis.

Dans cet exemple, l'intérêt est minime : il n'est pas très intéressant de vérifier que le tableau contient bien `1, 2, 3` car c'est comme cela qu'il a été initialisé. Mais `Data::Dumper` est d'une efficacité redoutable pour vérifier le contenu d'une structure construite dynamiquement.

```
use Data::Dumper;
$Data::Dumper::Indent = 0;
# construction de la table XOR
# des 3 premiers entiers.
my @table_de_xor;
foreach my $i (0..2) {
  foreach my $j (0..2) {
    $table_de_xor[$i][$j] = $i xor $j;
  }
}
```

```
say Dumper(\@array);

# affiche à l'écran :
# $VAR1 = [[0,0,0],[1,1,1],[2,2,2]];
```

Le résultat affiché n'est pas bon, ce n'est pas le résultat de l'opération *XOR* sur les entiers entre eux. L'erreur vient du fait que l'exemple de code utilise l'opérateur `xor` alors que c'est l'opérateur `^` qui effectue un *XOR* sur les entiers *bits à bits*.

Le code correct est le suivant :

```
use Data::Dumper;
# affichage plus concis
$Data::Dumper::Indent = 0;

# construction de la table XOR
# des 3 premiers entiers
my @table_de_xor;
foreach my $i (0..2) {
  foreach my $j (0..2) {
    $table_de_xor[$i][$j] = $i ^ $j;
  }
}

say Dumper(\@table_de_xor);

# affiche à l'écran :
# $VAR1 = [[0,1,2],[1,0,3],[2,3,0]];
```

Cet exemple montre l'utilité principale de `Data::Dumper` : vérifier qu'une structure de données est bien conforme au résultat escompté. Cet outil est très important lors du déboggage des programmes.

Utiliser les tables de hachage

Une table de hachage est une structure qui permet d'associer un scalaire (généralement une chaîne de caractères), qui s'appelle alors *clé*, à un autre scalaire (généralement une autre chaîne, un nombre, ou une référence sur d'autres structures) : c'est la *valeur*.

Il est très courant en français d'utiliser simplement le mot « hash », comme contraction de « table de hachage ».

Les tables de hachage sont plus connues dans d'autres langages sous le nom de « tableaux associatifs » ; qu'en Perl, ils sont souvent plus souples.

Le *sigil* d'une table de hachage est `%` :

```
my %hash;
```

L'opérateur flèche double

Avant de décrire les fonctionnalités des tables de hachage, il est important de s'attarder sur l'opérateur « flèche double ». L'opérateur => est en fait un remplaçant de l'opérateur virgule, mais il permet d'économiser l'utilisation des guillemets, et rend le code source plus lisible. Voici deux extraits de codes, qui sont équivalents, utilisés dans ce cas pour construire un tableau.

```
# sans l'utilisation de =>
my @array = ('pomme','poire','fraise','framboise');
# avec l'utilisation de =>
my array = (pomme => 'poire', fraise => 'framboise');
```

L'opérateur => est équivalent à l'opérateur virgule, sauf qu'il force l'argument gauche à être une chaîne. On dit qu'il évalue le paramètre de gauche « en contexte de chaîne ».

Créer une table de hachage

```
%hash = (. . .)
```

La manière la plus courante de créer une table de hachage est de le faire à partir d'une liste :

```
my %hash = (
   'michel', 25,
   'jean', 42,
   'christine', 37,
 );
```

Cette table de hachage associe un prénom à un âge. Michel a 25 ans, Jean 42, et Christine 37. Nous pouvons écrire la même table de hachage en utilisant l'opérateur => :

```
my %hash = (
   michel => 25,
   jean => 42,
   christine => 37,
 );
```

Il est indéniable que cette deuxième forme est plus concise et claire : l'économie des guillemets n'est pas négligeable, et la relation d'associativité y est mieux représentée grâce à la flèche, même si le code source n'est pas indenté proprement :

```
my %hash = ( michel => 25, jean => 42,
   ➥christine => 37 );
# est plus claire que :
my %hash = ( 'michel', 25, 'jean', 42,
   ➥'christine', 37 );
```

Une table de hachage est ainsi une table d'association, qui permet de relier deux types de valeurs. Le seul impératif est que ces valeurs soient des scalaires. Mais comme une référence est un scalaire, il est possible et facile d'associer une clé à des structures.

La table de hachage ci-dessous illustre le fait qu'une clé peut être une chaîne ou un nombre (qui sera interprété comme une chaîne[6]), et les valeurs peuvent être n'importe quel scalaire, tel que nombre, chaîne ou référence (sur tableau ou sur table de hachage).

```
my %hash = (
  nombre  => 12,
  chaîne  => "ceci est une chaîne",
  pairs   => [ 0, 2, 4, 6, 8 ],
  impairs => [ 1, 3, 5, 7, 9 ],
  tableau => \@array,
  fruits  => [ qw( pomme, poire, citron,
  ➥ fraise) ],
  42      => "réponse",
);
```

Accéder aux éléments d'une table de hachage

```
$hash{clé}
```

Créer une table de hachage n'est utile que s'il est aisé de manipuler ses éléments. Une manière d'accéder à un élément d'une table de hachage est de connaître sa clé :

6. En fait n'importe quel scalaire peut être utilisé, mais en dehors d'un nombre ou d'une chaîne, cela donnera des résultats surprenants et probablement inutilisables.

`$hash{clé}`. Il n'est pas nécessaire de mettre des guillemets autour de la clé s'il n'y a pas d'ambiguïté[7].

Un élément d'une table de hachage peut être utilisé pour récupérer une valeur ou pour l'assigner, et donc ajouter et modifier des valeurs :

```
my %hash = ( jean => 25);
# ajouter une association
$hash{michel} = 34;
# modifier une association
$hash{jean} = 26;
# afficher une valeur
say "jean a " . $hash{jean} . "ans.";
say "michel a " . $hash{michel} . "ans.";
# Ici, les guillemets sont obligatoires
$hash{'marie-claire'} = 16;
say 'Marie-claire ' .
  ( $hash{'marie-claire'} > 17 ? 'est' :
  ➥ "n'est pas")
. ' majeure';
```

Cette manière d'accéder à un élément permet de le réinitialiser, c'est-à-dire de le mettre à `undef` :

```
my %hash = (jean => 25);
# ajouter une association
$hash{michel} = 34;
# effacer une association
$hash{michel} = undef;
# peut s'écrire également
undef $hash{michel};
```

7. C'est-à-dire si la clé ne contient pas de caractère espace ou un opérateur, et si elle ne commence pas par un chiffre. Par exemple, `test`, `chat_1`, `r0b1n3t` sont valides. Mais `Marie Claire`, `jean-jacques` sont invalides.

Attention

La table de hash de l'exemple précédent contient toujours deux associations : `jean` et `michel`. La *valeur* de l'association `michel` est `undef`, mais la table de hachage contient bien deux entrées.

Supprimer un élément d'une table de hachage

```
delete $hash{clé}
```

`delete` permet d'enlever une association d'une table de hachage, et s'utilise très simplement :

```
my %hash = (jean => 25);
delete $hash{jean}
# le hash ne contient plus rien
```

Tester l'existence et la définition d'un élément

```
exists $hash{clé}
defined $hash{clé}
```

`exists` permet de savoir si, pour une clé donnée, il existe une association dans la table de hachage en question. `defined` renvoie vrai si la valeur de l'association est définie, faux dans les autres cas.

Attention

Il ne faut pas confondre existence d'une association, et le fait que la valeur soit définie. Voici un exemple de code qui illustre les deux concepts :

```
my %hash = ( jean => 25,
             julie => undef,
           );
exists  $hash{jean}; # renvoie vrai
defined $hash{jean}; # renvoie vrai

exists  $hash{julie}; # renvoie vrai
defined $hash{julie}; # renvoie faux

exists  $hash{marc}; # renvoie faux
defined $hash{marc}; # renvoie faux
```

`exists` représente l'existence d'une association (même vers une valeur non définie), alors que `defined` correspond au fait que la valeur pointée par la clé soit définie.

Utiliser des tranches de hashs

```
hash{a..b}
```

Comme pour les tableaux, il est possible de manipuler des tranches (*slice* en anglais) de tableaux de hachage. Il suffit pour cela d'utiliser non pas *une* clé, mais *une liste*[8] de clés pour spécifier plusieurs valeurs, qui sont alors renvoyées sous forme de liste.

Attention

Lors d'une utilisation d'une tranche de hash, le sigil du hash se transforme de `%` en `@`.

8. Ou un tableau.

```
my %hash = ( jean => 25,
             julie => undef,
             michel => 12,
           );

my @ages = @hash{'jean', 'michel'};
# @ages contient (25, 12)

my @ages = @hash{'julie', 'michel'};
# @ages contient (undef, 12)
```

Bien sûr, il est possible d'assigner plusieurs valeurs à la fois en utilisant une tranche :

```
my %hash = ( jean => 25 );
@hash{'julie', 'michel'} = (42, 12);

# %hash vaut maintenant :
#  jean => 25,
#  julie => 42,
#  michel => 12
```

Comme les clés sont spécifiées grâce à une liste ou un tableau, il est possible d'utiliser l'opérateur `qw()` :

```
my %hash = ( jean => 25 );
@hash{ qw(julie michel) } = (42, 12);
```

Il est bien sûr possible de créer un tableau de clés et de l'utiliser pour définir une tranche :

```
my @cles = qw(julie michel);
my %hash = ( jean => 25 );
@hash{ @cles } = (42, 12);
```

Obtenir la liste des clés

```
keys %hash
```

`keys` renvoie la liste des clés du hash :

```
my %hash = ( jean => 25,
             julie => undef,
             michel => 12,
           );
my @cles = keys %hash
# @cles vaut ('jean', 'julie', 'michel')
```

Attention

L'ordre des clés renvoyées est aléatoire, mais c'est le même que l'ordre des valeurs renvoyées par `values`.

Obtenir la liste des valeurs d'une table de hachage

```
values %hash
```

`values` renvoie la liste des valeurs du hash :

```
my %hash = ( jean => 25,
             julie => undef,
             michel => 12,
           );
my @valeurs = values %hash;
# @valeurs vaut (25, undef, 12)
```

Attention

L'ordre des valeurs renvoyées est aléatoire, mais c'est le même que l'ordre des clés renvoyées par `keys`.

Dumper un hash

De la même manière que `Data::Dumper` peut être utilisé pour dumper un tableau, ce module permet également d'explorer un hash, ou tout autre structure de données complexe.

```
use Data::Dumper;
$Data::Dumper::Indent = 0;
my %hash = ( nom => 'jean',
             age => 12,
             notes => [ 10, 13.5, 18 ]
           );
say Dumper(\%hash);

# affiche à l'écran :
# $VAR1 = {'nom' => 'jean',
           'notes' => [10,'13.5',18],
           'age' => 12};
```

Il est important de se rendre compte qu'une table de hachage ne stocke pas l'ordre des couples clé-valeur, pour des raisons de performances[9].

Boucler sur un hash avec `foreach`

```
foreach (. . .) {. . .}
```

La première méthode classique pour parcourir une table de hachage est d'utiliser `foreach` sur les clés du hash, et de récupérer la valeur pour chaque clé :

9. Il existe des modules qui proposent des tables de hachage qui gardent l'ordre, ou même qui trient à la volée leur contenu, comme par exemple `Tie::Hash::Sorted`.

```
my %hash = ( jean => 25,
             michel => 12,
           );
foreach my $key (keys %hash) {
  my $value = $hash{$key};
  say "l'age de $key est : $value";
}
```

Cette méthode a l'avantage d'être explicite et simple à comprendre.

Boucler sur un hash avec each

each

Il existe cependant un autre moyen de « boucler » à la fois sur les clés et valeurs d'une table de hachage, en utilisant la fonction each :

```
my %hash = ( jean => 25,
             michel => 12,
           );
while (my ($key, $value) = each %hash) {
  say "l'age de $key est : $value";
}
```

La fonction each renvoie les couples clé-valeur du hash un à un. Lorsque tous les couples ont été renvoyés, each renvoie une liste vide (c'est ce qui fait que la boucle while s'arrête), puis recommence au début du hash.

Attention

L'ordre des couples clé-valeur renvoyés est aléatoire, mais c'est le même que l'ordre des clés renvoyées par keys ou celui des valeurs renvoyées par values.

Référencer un hash

```
{ clé => valeur } \%hash
```

Comme pour les tableaux, il est aisé de créer une référence sur un hash.

Pour obtenir une référence, il est possible d'utiliser \ sur une table de hachage, qui renvoie une référence pointant dessus. Pour créer une référence sur un ensemble de clé-valeur directement, il existe également `{ }` :

```
my %hash = ( nom => "Hugo", prenom =>
   ➥"Victor" );
my $ref_hash = \%hash;
# $ref_hash est une référence sur @hash

my $ref_hash = { nom => "Hugo", prenom =>
   ➥"Victor" };
# $ref_hash est une référence
```

Déréférencer un hash

```
%{$ref_array}
```

Pour accéder à un hash référencé, il faut utiliser `%{}` :

```
my $ref_hash = { nom => "Hugo", prenom =>
   ➥"Victor" };
my %hash = %{$ref_hash};
```

L'opérateur `%{}` évalue l'expression donnée, et déréférence le résultat en hash. Lorsqu'il n'y a pas d'ambiguïté, les accolades peuvent être omises :

```
my $ref_hash = { nom => "Hugo", prenom =>
   ➥"Victor" };
my %hash = %$ref_hash;
```

Pour accéder directement à un élément d'un hash à partir d'une référence, il faut utiliser l'opérateur flèche -> :

```
my $ref_hash = { nom => "Hugo", prenom =>
   ➥"Victor" };
my $prenom = $ref_hash->{prenom};
# $prenom vaut "Victor"
```

Créer des structures hybrides

Il est facile de créer des structures plus complexes, en combinant les tableaux, hash, listes et en utilisant des références.

Pour créer un hash de tableaux, il suffit d'utiliser des références de tableaux, et de les stocker dans une table de hachage. Ainsi, il est facile de créer une structure de données pour représenter un élève et ses notes sur les trois premiers mois :

```
my %eleve1 = (
   nom => 'Dupont',
   prenom => 'Jean',
   notes => { janvier => [ 12, 9, 14, 10 ],
              fevrier => [ 13, 15, 8, 9 ],
              mars => [ 12, 8, 11, 12.5 ]
            }
           );
```

Accéder à la troisième note du mois de février de cet élève se fait ainsi :

```
my $note = $eleve1{notes}->{février}->[2];
# note vaut 8
```

Ce code peut également s'écrire comme suit :

```
my $note = $eleve1{notes}{février}[2];
```

Transformer tous les éléments d'un tableau ou d'une liste avec `map`

```
map {. . .} @tableau
```

Les opérations de base sur les tableaux et listes consistent en général à manipuler un ou plusieurs éléments. Mais Perl propose de manipuler des listes en entier grâce à `map` et `grep`.

`map` permet d'appliquer des transformations sur tous les éléments d'un tableau ou d'une liste. Il prend en argument :

- un bloc de code[10] à exécuter ;
- une liste ou un tableau.

Le bloc de code sera exécuté autant de fois qu'il y a d'éléments dans la liste ou le tableau, et pour chaque itération, la variable spéciale `$_` prendra la valeur de l'élément.

```
# exemple avec un tableau
my @tableau = (1..3);
@tableau2 = map { $_ * 2 } @tableau1;
# @tableau2 contient ( 2, 4, 6 )

# même exemple avec une liste
@tableau = map { $_ * 2 } (1..3);
# @tableau contient ( 2, 4, 6 )
```

10. Il est possible de donner une expression comme premier argument, mais cette manière d'utiliser `map` est déconseillée.

Il est possible d'écrire l'équivalent d'un code utilisant `map` avec un morceau de code utilisant `foreach`, mais généralement cela prend plus de lignes. Voici un exemple qui illustre très bien cela. Il s'agit d'écrire une fonction qui reçoit en argument une liste de chaînes, et qui renvoie une nouvelle liste contenant les chaînes mises en majuscules.

```
# exemple avec foreach
sub majuscules {
  my @chaines = @_;
  my @resultat;
  foreach my $chaine (@chaines) {
    push @resultat, uc($chaine);
  }
  return @resultat;
}

# exemple avec map
sub majuscules {
  my @chaines = @_;
  return map { uc } @chaines;
}
```

Il convient d'expliquer ici que :

- La fonction `uc` prend en argument une chaîne de caractères et la met en majuscule. Dans argument, elle utilise la variable spéciale `$_`.
- `map` renvoie la liste résultat (sans modifier la liste originelle). Or il ne sert à rien de stocker le résultat dans un tableau intermédiaire, autant directement le renvoyer avec `return`.

Il faut bien comprendre que `map` construit une nouvelle liste avec les résultats du bloc de code exécuté. Le bloc de code n'est donc pas obligé de renvoyer toujours un résultat, et il peut également en renvoyer plus d'un. Voici un exemple qui supprime les nombres inférieurs à 4 d'une liste, et pour tout autre nombre, il insère son double à la suite.

```
my @list = map {
  $_ >= 4          # si le nombre est > 4
    ? ($_, 2*$_)   # le nombre et son double
    : ()           # sinon rien
} (1..5);
# @liste contient (4, 8, 5, 10)
```

Astuce

`map` est principalement utilisé pour créer des tableaux, mais il peut également être utilisé pour créer des hashs. En effet, un hash s'initialise grâce à une liste, comme dans le code suivant :

```
my %hash = ( nom => "Jean" );
# équivalent à
my %hash = ( "nom", "Jean" );
```

Dans cet exemple, le hash `%hash` est initialisé grâce à une liste de deux chaînes, la première correspondant au nom de la clé, la deuxième à la valeur. Cette liste peut parfaitement être la valeur de retour d'une fonction, et donc `map` peut être utilisé pour créer cette liste :

```
my $count = 0;
my %hash = map {
    "eleve_" . $count++ => $_;
} ( qw(Jean Marcel) );

# %hash contient :
# eleve_0 => "Jean"
# eleve_1 => "Marcel"
```

Filtrer un tableau ou une liste avec `grep`

```
grep {. . .} @tableau
```

C'est le compagnon de `map`. `grep` permet de filtrer un tableau ou une liste, et renvoie une liste résultat ne contenant que les éléments qui passent un test donné. En exemple, voici comment filtrer les nombres d'une liste pour ne garder que ceux inférieurs à 10 :

```
my @resultat = grep { $_ < 10 } @liste;
```

Voici un extrait de code qui élimine tous les nombres impairs d'une liste de nombres :

```
my @liste = (1..5);
my @resultat = grep { ! $_ % 2 } @liste
```

L'opérateur `%` calcule le *modulo* de deux nombres. `$_ % 2` renverra donc zéro si `$_` est pair. Pour garder les nombres pairs, il faut que le test soit vrai ; il donc faut inverser le test avec `!`.

Info

En plus des opérations classiques sur les listes, Perl propose un assortiment de fonctions plus puissantes, qui ont été elles-même complétées par un nombre important de modules disponibles sur CPAN. Seule une sélection de deux modules éprouvés et stables est présentée ici, `List::Util` et `List::MoreUtils`.

Renvoyer le premier élément d'une liste avec `List::Util`

```
first(..)
```

Le module `List::Util` (disponible en standard avec Perl) contient quelques méthodes pour travailler sur les listes, afin d'étendre les opérateurs de base du langage.

first renvoie le premier élément d'une liste ou d'un tableau.

```
use List::Util qw(first);
my $resultat = first(1..3);
# $resultat contient 1
```

Trouver le plus grand élément avec List::Util

```
max(..)
maxstr(..)
```

max renvoie le plus grand élément, en les comparant numériquement.

```
use List::Util qw(max);
my $resultat = max(1, 3, 2, 5, -2);
# $resultat contient 5
```

maxstr renvoie l'élément qui a la plus grande valeur alphabétique.

```
use List::Util qw(maxstr);
my $resultat = maxstr('quelques','chaines');
# $resultat contient 'quelques';
```

Trouver le plus petit élément avec List::Util

```
min(..)
minstr(..)
```

`min` renvoie le plus petit élément d'une liste, en les comparant numériquement.

```
use List::Util qw(min);
my $resultat = min(1, 3, 2, 5, -2);
# $resultat contient -2
```

`minstr`, quant à elle, renvoie l'élément qui a la plus petite valeur alphabétique.

```
use List::Util qw(maxstr);
my $resultat = maxstr('quelques','chaines');
# $resultat contient 'chaines';
```

Réduire une liste avec List::Util

```
reduce {. . .} (. . .)
```

Cette méthode réduit une liste, en appliquant un bloc de code qui reçoit deux argument `$a` et `$b`. Lors de la première évaluation, ces deux variables sont initialisées aux deux premiers éléments de la liste. Lors des évaluations ultérieurs, `$a` vaut le résultat de l'évaluation précédente, et `$b` a comme valeur le prochain élément de la liste.

```
use List::Util qw(reduce);
$resultat = reduce { $a * $b } (1..10)
# $resultat contient le produit
# des nombres de  1 à 10
```

Mélanger une liste avec List::Util

```
shuffle(..)
```

Il est aisé de mélanger une liste de manière pseudo-aléatoire avec cette fonction, qui s'utilise très facilement :

```
use List::Util qw(reduce);
my @tableau = (1..10);
my @melange = shuffle(@tableau);
```

Faire la somme d'une liste avec List::Util

```
sum(..)
```

Comme son nom l'indique, sum sert simplement à faire la somme de tous les éléments d'une liste :

```
use List::Util qw(sum);
my $somme = (1..100);
# somme vaut 5050
```

Savoir si un élément vérifie un test avec List::MoreUtils

```
any {. . .} @tableau
```

La fonction any renvoie vrai si au moins un des éléments d'une liste vérifie un test donné.

Savoir si aucun élément ne vérifie un test avec List::MoreUtils

```
none {. . .} @tableau
```

La fonction `none` renvoie vrai si aucun des éléments d'une liste ne vérifie un test donné.

Appliquer du code sur deux tableaux avec List::MoreUtils

```
pairwise {. . .} (@tableau1, @tableau2)
```

Cette fonction très utile permet d'exécuter un bloc de code sur deux tableaux à la fois : dans le bloc de code, les variables `$a` et `$b` prennent successivement les premiers des éléments des deux tableaux respectivement, plus les deuxièmes éléments, etc. jusqu'à la fin de la liste. `pairwise` renvoie la liste de résultats.

Tricoter un tableau avec List::MoreUtils

```
zip(@tableau1, @tableau2, . . .)
```

`zip` prend en argument un nombre quelconque de tableaux, et renvoie une liste constituée du premier élément de chaque tableau, puis du deuxième élément de chaque tableau, etc., jusqu'à la fin de tous les tableaux.

Enlever les doublons avec `List::MoreUtils`

```
uniq(@tableau)
```

Cette fonction renvoie une liste contenant les éléments du tableau passé en paramètre, mais sans les doublons.

Info

L'ordre initial des éléments est respecté dans la liste résultat.

5

Expressions régulières

L'une des forces traditionnelles de Perl réside dans ses capacités d'analyse, avec en premier lieu ce véritable langage dans le langage que sont les expressions régulières. Celles-ci permettent de comparer une chaîne à un motif, pour vérifier si elle correspond et éventuellement en extraire des parties intéressantes.

Si leur syntaxe apparaît assez cryptique de prime abord, c'est parce qu'il s'agit avant tout d'un langage très spécialisé, dédié à une tâche particulière, la recherche de correspondance de motifs. Larry Wall, le concepteur de Perl, s'amusait d'ailleurs que bien que les expressions régulières soient en partie à l'origine de la réputation de mauvaise lisibilité de Perl, les autres langages de programmation se sont empressés de les emprunter pour eux-mêmes.

Qu'est-ce qu'une expression régulière ?

Tout d'abord, un petit peu de théorie et d'histoire. Les expressions régulières (certains préfèrent parler d'*expressions rationnelles*) sont une grammaire s'inscrivant dans le cadre de la théorie des langages formels, qui décrit comment analyser et

exécuter de tels langages au travers d'automates à états finis. Ces derniers se distinguent en automates à états finis déterministes ou DFA (*Deterministic Finite Automaton*), s'il est possible de calculer si l'automate s'arrêtera et à quel moment, et non déterministes ou NFA (*Non-deterministic Finite Automaton*) dans le cas contraire.

Ken Thompson, l'un des pères du système Unix, écrivit les premières implémentations d'expressions régulières, de type DFA, dans les éditeurs de lignes *QED* et *ed*, puis dans l'utilitaire *grep* en 1973. La syntaxe d'origine (dite SRE) fut par la suite augmentée et normalisée au sein de POSIX sous forme de deux syntaxes différentes (BRE et ERE). Une implémentation de ces expressions régulières POSIX, écrite par Henry Spencer, servit de base pour celle de Perl, qui étendit et surtout rationalisa de manière très importante la grammaire sous-jacente. C'est elle qui fait maintenant office de quasi-standard, au travers de sa réimplémentation PCRE (*Perl Compatible Regular Expressions*) écrite par Philip Hazel pour le serveur de courriels Exim, et qui est maintenant utilisée par de très nombreux logiciels.

Effectuer une recherche

```
m//
```

`m//` est l'opérateur de recherche (*match*), qui prend comme argument une expression régulière : `m/REGEXP/`. Le délimiteur par défaut est la barre oblique (`/`), mais il peut être changé par n'importe quel caractère (hormis l'espace et le saut de ligne, pour d'évidentes raisons de lisibilité), et en particulier par les couples de caractères suivants : `m(..)`,

`m[..]`, `m{..}`, `m<..>`. Ces derniers sont très pratiques car Perl reconnaît correctement les imbrications des caractères identiques faisant partie de la syntaxe des expressions, et ils permettent généralement une meilleure présentation pour les expressions complexes sur plusieurs lignes.

Quand le délimiteur est celui par défaut, le `m` initial peut être omis, d'où la forme abrégée `//` qui va être utilisée très souvent dans ce chapitre.

Par défaut, `m//` travaille sur la variable `$_`, mais il suffit d'utiliser l'opérateur de liaison `=~` pour travailler sur une autre variable :

```
$string =~ /ab/;
# test de $string contre le motif "ab"
```

Le motif peut contenir des variables, qui seront interpolées (et le motif recompilé) à chaque évaluation.

En contexte scalaire, `m//` renvoie vrai si le texte correspond au motif, faux sinon. En contexte de liste, il renvoie la liste des groupes capturants (`$1`, `$2`, `$3`, etc.) en cas de succès, et la liste vide sinon.

Rechercher et remplacer

`s///`

`s///` est l'opérateur de recherche et remplacement, qui prend comme arguments une expression régulière et une chaîne de remplacement : `s/REGEXP/REMPLACEMENT/`. Comme pour `m//`, les délimiteurs peuvent être changés, avec le petit raffinement supplémentaire qu'il est possible d'utiliser des paires différentes pour les deux parties de l'opérateur : `s{..}<..>`.

`s///` travaille lui aussi par défaut sur la variable `$_`, et comme précédemment il suffit d'utiliser l'opérateur de liaison =~ pour travailler sur une autre variable :

```
# cherche la chaîne "ab" et la remplace
# par "ba"
$string =~ s/ab/ba/;
```

Là encore, le motif interpole les éventuelles variables qu'il contient. La chaîne de remplacement fonctionne véritablement comme une chaîne normale, et supporte donc sans problème l'interpolation des variables, y compris celles qui contiennent les valeurs des groupes capturants (`$1`, `$2`, `$3`, etc.).

En plus des modificateurs décrits ci-après, reconnus tant par `m//` que par `s///`, ce dernier en supporte un spécifique, `/e`, qui évalue la chaîne de remplacement avec la fonction `eval`. Ce n'est pas très performant, mais cela permet de réaliser simplement certaines opérations, par exemple le petit bout de code suivant permet de remplacer tous les caractères encodés dans les URL par leur vraie valeur :

```
$string =~ s/%([A-Fa-f0-9]{2})/chr(hex($1))
   ➥/eg;
```

Il est même possible d'évaluer le résultat obtenu en ajoutant un autre modificateur `/e`.

`s///` renvoie le nombre de remplacements effectués s'il y en a eu, faux sinon.

Stocker une expression régulière

```
qr//
```

Une expression régulière peut être stockée dans une variable scalaire en utilisant la syntaxe `qr//` :

```
my $re = qr/ab/;
```

Comme tous les q-opérateurs, il est possible de changer le caractère délimitant la chaîne de l'opérateur. Les expressions suivantes définissent la même expression régulière que ci-dessus :

```
my $re = qr{ab};
my $re = qr!ab!;
```

Il est alors possible de tenter une correspondance avec un texte avec l'opérateur de liaison =~ vu page 96 :

```
$text =~ $re;
```

Rechercher sans prendre en compte la casse

```
/i
```

Info

Une recherche de correspondance peut être affinée à l'aide de *modificateurs*, qui s'écrivent sous la forme de simples lettres en fin d'opérateur. Plusieurs modificateurs peuvent se combiner sans problème d'incompatibilité.

Par défaut, la recherche s'effectue en respectant la casse (la différence entre majuscules et minuscules). Quand ce modificateur est activé, la recherche devient insensible à la casse :

```
"BAM" =~ /bam/;      # ne correspond pas
"BAM" =~ /bam/i;     # correspond
```

Rechercher dans une chaîne multiligne

```
/m
```

Ce modificateur fait considérer la chaîne comme contenant plusieurs lignes. En réalité, cela change le comportement des métacaractères ^ et $ (voir page 102) pour qu'ils correspondent respectivement au début et fin des lignes au sein de la chaîne.

Rechercher dans une chaîne simple

```
/s
```

Ce modificateur fait considérer la chaîne comme ne comprenant qu'une seule ligne. En réalité, cela change le comportement du métacaractère point . pour qu'il corresponde aussi au saut de ligne.

Malgré leur description qui semble contradictoire, les modificateurs /m et /s peuvent se combiner ensemble sans problème : quand /ms est utilisé, cela modifie le métacaractère point . pour qu'il corresponde à tout caractère, y compris le saut de ligne, tout en autorisant les métacaractères ^ et $ à correspondre après et avant les sauts de lignes embarqués.

Neutraliser les caractères espace

```
/x
```

Le modificateur de syntaxe retire toute signification aux caractères d'espace, qui doivent alors être explicitement écrits sous forme de métacaractères pour correspondre. Cela permet d'écrire les expressions régulières sur plusieurs lignes, avec des commentaires, ce qui est extrêmement pratique pour les motifs longs et complexes :

```
$input =~ m{
  ^               # début de ligne
  ((?:(?:(?:dis¦en)abl¦pag)e¦status))
                  # commande ($1)
  (\+\d+)?        # le décalage ($2)
  \s+             # des espaces
  (\S+?)\.(\S+)   # serveur.sonde ($3, $4)
  \s+             # des espaces
  (.*)$           # derniers arguments ($5)
}sx;
```

Exemple tiré d'un module CPAN qui permet l'analyse des messages au protocole Big Brother/Hobbit.

Contrôler la recherche globale de correspondances

```
/g, /c, /gc
```

Le modificateur de correspondance globale `/g` active la recherche globale de correspondances, c'est-à-dire que le motif va parcourir la chaîne au fur et à mesure des corres-

pondances réussies. Pour ce faire, la position courante de travail est normalement remise à zéro après échec, sauf si le modificateur `/c` est activé.

Distinguer caractères normaux et métacaractères

Un motif est constitué d'éléments appelés « atomes » dans la terminologie reconnue. Les plus simples atomes d'un motif sont les caractères normaux, qui établissent une correspondance avec eux-mêmes.

```
"Hello world" =~ /World/; # correspond
"Hello world" =~ /lo Wo/; # correspond aussi
```

Il est très important de noter qu'un motif va toujours s'arrêter à la *première correspondance* trouvée :

```
# le "hat" de "That" correspond
"That hat is red" =~ /hat/;
```

Attention

Cela surprend souvent les débutants, mais le langage des expressions régulières ne permet pas naturellement de récupérer la *n*-ième occurrence d'une correspondance. Bien sûr, cela peut se contourner, de manière interne pour les cas simples, ou de manière externe dans les cas plus complexes.

Par défaut, la casse (la distinction majuscule-minuscule) est prise en compte, et les espaces (espace normale, tabulation, retour chariot) sont considérées comme des caractères normaux.

Plus précisément, est considéré comme caractère normal tout caractère qui n'est pas un métacaractère, ces derniers

étant utilisés pour la syntaxe des expressions régulières :
`{ } [ ] ( ) ^ $ . | * ? + \`

Tout métacaractère redevient un caractère normal en le préfixant par une barre oblique inverse \ (*antislash* ou *backslash* en anglais), ainsi :

```
"2+2=4" =~ /2+2/;    # ne correspond pas
"2+2=4" =~ /2\+2/;   # ok, correspond
```

Bien sûr, le caractère utilisé pour délimiter l'expression régulière doit lui-même être protégé afin de pouvoir être utilisé, d'où l'intérêt de pouvoir changer le délimiteur, afin d'éviter le syndrome de la barre oblique :

```
"/usr/bin/perl" =~ /\/usr\/bin\/perl/;
"/usr/bin/perl" =~ m{/usr/bin/perl};
```

Inversement, un caractère normal peut devenir un métacaractère en le préfixant par un antislash. Certains introduisent des séquences d'échappement, c'est-à-dire qu'ils prennent en argument une séquence de caractères normaux. Pour une part, il s'agit en fait simplement de métacaractères déjà bien connus : `\t` pour la tabulation, `\n` pour une nouvelle ligne, `\xHH` pour obtenir un caractère par son code point en hexadécimal, etc.

```
# correspond aussi (même si c'est bizarre)
"cat" =~ /\143\x61\x74/;
```

Les autres caractères, qui ne sont pas normaux, font partie des classes de caractères (voir la section suivante).

Établir une correspondance parmi plusieurs caractères

Une *classe de caractères* est un atome de longueur un qui permet d'établir une correspondance parmi plusieurs caractères possibles. Elle est notée par des crochets `[...]` qui encadrent les valeurs possibles. Ainsi `/[bcr]at/` peut correspondre aux chaînes « bat », « cat » et « rat ».

Comme il serait fastidieux d'écrire l'ensemble des lettres, il est possible de définir des intervalles de caractères en mettant un tiret « - » entre les deux bornes. Ainsi `[0-9]` est équivalent à `[0123456789]`, et `[b-n]` à `[bcdefghijklmn]`. L'expression `/[0-9a-fA-F]` permet donc de chercher un caractère hexadécimal.

Dernier point de la syntaxe des classes de caractères, la négation d'une classe, qui s'active en préfixant par l'accent circonflexe `^`. `/[^0-9]/` correspond à tout caractère qui n'est pas un chiffre.

Comme certaines classes sont très courantes, elles sont prédéfinies dans les expressions régulières. La plus simple est le métacaractère point `.` qui correspond à tout caractère sauf le saut de ligne :

```
"a=3" =~ /.../;      # correspond
```

Il est bon de se souvenir que cela peut se modifier avec le modificateur `/s` :

```
"krack\nboum"=~ /krack.boum/; # ne corr. pas
"krack\nboum"=~ /krack.boum/s; # correspond
```

Il existe deux syntaxes pour utiliser les autres classes prédéfinies. L'une, introduite par Perl, est courte et correspond aux métacaractères qui ont été laissés en suspens, de la

forme antislash plus un caractère minuscule, avec la forme majuscule qui correspond à sa négation :

- `\w` (comme *word character*) correspond à un caractère de « mot », c'est-à-dire tout alphanumérique plus l'espace soulignée (_), et `\W` à tout caractère qui n'est pas un mot.
- `\d` (comme *digit*) correspond à un chiffre, et `\D` à tout ce qui n'est pas un chiffre :

```
/\d\d:\d\d:\d\d/;
# format d'heure hh:mm:ss
```

- `\s` (comme *space*) correspond à tout caractère considéré comme un espace blanc (fondamentalement, la tabulation, l'espace usuelle et les sauts de ligne et de page), et `\S` à tout caractère qui n'est pas considéré comme un espace blanc.

L'autre syntaxe, dite POSIX, est de la forme `[:classe:]`, à utiliser au sein d'une classe, dont la syntaxe complète est `[[:classe:]]`. Perl permet de plus de créer les négations de ces classes par `[^[:classe:]]`. La liste des classes reconnues est :

- `[[:alpha:]]` correspond à tout caractère alphabétique, par défaut `[A-Za-z]` ;
- `[[:alnum:]]` correspond à tout caractère alphanumérique, par défaut `[A-Za-z0-9]` ;
- `[[:ascii:]]` correspond à tout caractère dans le jeu de caractères ASCII ;
- `[[:blank:]]` une extension GNU qui correspond à l'espace usuelle et à la tabulation ;
- `[[:cntrl:]]` correspond à tout caractère de contrôle ;
- `[[:digit:]]` correspond à tout caractère considéré comme un chiffre, équivalent à `\d` ;
- `[[:graph:]]` correspond à tout caractère imprimable, à l'exclusion des espaces ;

- `[[:lower:]]` correspond à tout caractères minuscule, par défaut `[a-z]` ;
- `[[:print:]]` correspond à tout caractère imprimable, y compris les espaces ;
- `[[:punct:]]` correspond à tout caractère de ponctuation, c'est-à-dire tout caractère imprimable à l'exclusion des caractères de mot ;
- `[[:space:]]` correspond à tout caractère d'espace, équivalent à `\s` plus la tabulation verticale `\cK` ;
- `[[:upper:]]` correspond à tout caractère majuscule, par défaut `[A-Z]` ;
- `[[:word:]]` est une extension Perl équivalente à `\w` ;
- `[[:xdigit:]]` correspond à tout caractère hexadécimal.

Il faut noter que, au sein d'une classe de caractères, la plupart des métacaractères de la syntaxe des expressions régulières perdent leur signification, hormis ceux qui correspondent à des caractères (comme `\t` et `\n`) et ceux qui correspondent à des classes (comme `\d`).

Ancrer une expression régulière en début de ligne

```
^
```

Info

Les *ancres* sont des assertions de longueur nulle qui permettent, comme leur nom l'indique, d'ancrer la recherche en certains points de la chaîne, afin d'imposer des contraintes au motif pour, par exemple, limiter ou au contraire étendre sa portée d'exécution.

L'ancre ^ permet d'accrocher le motif en début de ligne :

```
"beausoleil" =~ /^soleil/; # ne corresp. pas
"beausoleil" =~ /^beau/;   # correspond
```

Par défaut, ^ garantit de ne correspondre qu'en début de chaîne, y compris si celle-ci comporte des sauts de ligne internes. Pour que cette ancre corresponde en plus après chaque saut de ligne, il suffit d'utiliser le modificateur /m :

```
"un\ndeux\ntrois" =~ /^deux/m;  # correspond
```

Ancrer une expression régulière en fin de ligne

```
$
```

L'ancre $ permet d'accrocher le motif en fin de ligne :

```
"beausoleil" =~ /beau$/;   # ne corresp. pas
"beausoleil" =~ /soleil$/; # correspond
```

Comme pour ^, $ garantit par défaut de ne correspondre qu'en fin de chaîne, et il faut utiliser le modificateur /m pour qu'elle corresponde en plus avant chaque saut de ligne interne :

```
"un\ndeux\ntrois" =~ /deux$/m;  # correspond
```

Utiliser une ancre de début de chaîne

```
\A
```

\A est comme ^ une ancre de début de chaîne, sauf qu'elle n'est pas affectée par le modificateur /m.

Utiliser une ancre de fin de chaîne

```
\Z, \z
```

\Z est comme $ une ancre de fin de chaîne, sauf qu'elle n'est pas affectée par le modificateur /m. Comme \Z peut tout de même correspondre juste avant un saut de ligne final, il existe aussi \z qui garantit de ne correspondre qu'à la fin physique de la chaîne.

Utiliser une ancre de frontière de mot

```
\b
```

L'ancre \b (*boundary*) est conçue pour correspondre à la frontière d'un mot, qui intervient entre un \w et un \W.

```
"beausoleil" =~ /\bbeau/;    # correspond
"beausoleil" =~ /\bbeau\b/; # ne corres. pas
"beausoleil" =~ /\bsoleil/; # ne corres. pas
```

Utiliser une ancre par préfixe de recherche

```
\K
```

L'ancre `\K` (comme *keep*), introduite avec Perl 5.10 et auparavant par le module CPAN `Regexp::Keep`, permet de positionner la recherche de correspondance en un point précis de la chaîne examinée. Plus précisément la partie à gauche de cette assertion est un motif de recherche pure, utilisé seulement pour le positionnement. La principale utilité de cette assertion est d'offrir une solution plus efficace à un cas courant d'utilisation de `s///`, l'effacement d'un morceau de texte repéré par un préfixe particulier qui est à conserver :

```
$text =~ s/(préfixe)texte à supprimer/$1/;
```

se réécrit :

```
$text =~ s/préfixe\Ktexte à supprimer//;
```

qui est un peu plus clair, mais est surtout deux fois plus rapide à s'exécuter.

Utiliser une ancre de correspondance globale

```
\G pos()
```

L'ancre `\G` ne correspond qu'à la position courante du motif dans la chaîne, qui peut s'obtenir et se modifier par la fonction `pos()`.

```
$str = "abcdef";
$str =~ /cd/g;
say pos($str);     # affiche 4
```

Cette ancre s'utilise typiquement sous la forme idiomatique `m/\G. . ./gc` pour écrire des analyseurs bas niveau de type *lex*. Il n'est pas utile de s'attarder davantage sur ce point car il existe des modules CPAN permettant de réaliser des analyses syntaxiques et grammaticales de manière plus aisée. Par exemple `Parse::Yapp` (un peu vieux mais toujours utile) qui offre un fonctionnement de type *yacc*, et `Marpa` qui supporte des grammaires de type EBNF. Également les deux modules de Damian Conway, `Parse::RecDescent` et le récent `Regexp::Grammars` qui rend obsolète le premier et permet d'écrire des grammaires arbitrairement complexes.

Unicode et locales

Il n'est pas possible d'aborder ici en détail les problèmes liés à Unicode car il faudrait un livre entier dédié à ce sujet pour le couvrir de manière correcte ; voici cependant un rappel des bases.

Unicode est une norme qui constitue un catalogue des centaines de milliers de caractères des écritures du monde entier, de leurs propriétés associées et de leurs règles d'utilisation. Au-delà de notre alphabet romain si familier, et même des alphabets proches comme le grec ou le cyrillique, il existe de nombreux systèmes d'écritures qui obéissent à des règles totalement différentes. Par exemple les systèmes hébreu et arabe s'écrivent de droite à gauche. Les caractères arabes ont différentes variantes en fonction de leur position dans un mot. Les syllabaires japonais et les idéogrammes chinois n'ont pas de

notion de casse. Plusieurs systèmes d'écritures possèdent leurs propres définitions de chiffres. Sans compter les centaines de symboles techniques et mathématiques.

Le moteur d'expressions régulières de Perl supporte sans problème l'analyse de chaînes en Unicode (dans le format interne de Perl qui est une variation de UTF-8), mais ce support implique des comportements, spécifiés dans la norme Unicode, qui peuvent surprendre l'utilisateur non averti. Ceux-ci s'activent typiquement par le chargement des locales qui changent certaines définitions comme les caractères qui constituent un mot (`\w`), les caractères considérés comme des chiffres (`\d`), et même les caractères considérés comme des espaces (`\s`). Ainsi, si la locale française va inclure les caractères diacritiques (é, à, ù, etc.) dans les caractères de mot (`\w`), ce n'est pas le cas d'une locale anglaise.

En résumé, devant un comportement curieux avec une expression régulière, il est souvent utile de vérifier l'environnement et le type des données sur lesquelles l'expression est testée.

Quantifieur *

```
x*
```

Info

Les *quantifieurs* permettent la répétition d'un sous-motif, pour que celui-ci établisse une correspondance multiple de ses atomes avec la chaîne examinée.

Le quantifieur `*` établit une correspondance de zéro ou plusieurs fois :

```
"krack"      =~ /kra*ck/;     # correspond
"kraaack"    =~ /kra*ck/;     # correspond
"krck"       =~ /kra*ck/;     # correspond
```

Le dernier exemple montre bien que le sous-motif peut réussir en correspondant zéro fois.

Quantifieur +

```
x+
```

Le quantifieur + établit une correspondance de une ou plusieurs fois :

```
"krack"      =~ /kra+ck/;     # correspond
"kraaack"    =~ /kra+ck/;     # correspond
"krck"       =~ /kra+ck/;     # ne corres. pas

"item04"     =~ /\w+/;        # correspond
"Kevin68"    =~ /\w+/;        # correspond
"srv-http"   =~ /\w+/;        # ne corres. pas

# recherche simpliste d'une adresse IPv4
/\d+\.\d+\.\d+\.\d+/;

# recherche simpliste d'une adresse mail
/<[-.\w]+\@[-.\w]+>/;
```

Quantifieur ?

```
x?
```

Le quantifieur ? établit une correspondance de zéro ou une fois, c'est-à-dire qu'il rend le sous-motif optionnel :

```
"krack"      =~ /kra?ck/;    # correspond
"kraaack"    =~ /kra?ck/;    # ne corres. pas
"krck"       =~ /kra?ck/;    # correspond
```

Quantifieur {n}

```
x{n}
```

Le quantifieur `{n}` établit une correspondance d'exactement *n* fois :

```
"krack"    =~ /kra{2}ck/;   # ne corresp. pas
"kraack"   =~ /kra{2}ck/;   # correspond
"kraaack"  =~ /kra{2}ck/;   # ne corresp. pas
```

Quantifieur {n,m}

```
x{n,m}
```

Le quantifieur `{n,m}` établit une correspondance entre *n* et *m* fois :

```
"krack"    =~ /kra{2,4}ck/; # ne corr. pas
"kraaack"  =~ /kra{2,4}ck/; # correspond
"kraaaaack" =~ /kra{2,4}ck/; # ne corr. pas
```

La borne supérieure *m* peut ne pas être précisée, auquel cas la correspondance essayera de s'établir au moins *n* fois :

```
"kraaack"    =~ /kra{5,}ck/; # ne corres. pas
"kraaaaack"  =~ /kra{5,}ck/; # correspond
```

Quantifieurs non avides

```
x*? x+? x*?? x{n}? x{n,m}?
```

Un point important à noter est que les quantifieurs précédents sont dits *avides* et tentent d'établir la correspondance la plus grande possible avec les caractères de la chaîne cible. Voici un exemple typique des débutants :

```
"<a href="..."><img src="..."></a>"
   ➥=~ /<.+>/;
```

Tout d'abord, essayer d'analyser du HTML (ou du XML) avec des expressions régulières, c'est « mal et ça ne marche pas ». Dans ce cas-ci, l'utilisateur voulait capturer le premier élément `<a>`, mais le comportement avide de l'expression fait que c'est en réalité l'ensemble des caractères jusqu'au `>` du `</a>` qui sont pris en compte.

Il existe donc une variation des quantifieurs qui permet de les rendre *non avides*, pour qu'ils tentent d'établir une correspondance au plus tôt :

- `*?` zéro ou plusieurs fois, au plus tôt ;
- `+?` une ou plusieurs fois, au plus tôt ;
- `??` zéro ou une fois, au plus tôt ;
- `{n}?` exactement *n* fois, au plus tôt ;
- `{n,}?` au moins *n* fois, au plus tôt ;
- `{n,m}?` entre *n* et *m* fois, au plus tôt.

L'exemple précédent, corrigé pour devenir non avide, devient `/<.+?>/`. Il est aussi possible d'utiliser une classe qui exclut le caractère terminal, donc ici `/<[^>]+>/`.

Il faut se souvenir que là, plus encore qu'avant, le motif va terminer dès qu'une correspondance est trouvée.

```
"aaaa"     =~ /a+?/;     # correspond avec "a"
"aaaabbbb"=~ /a+?b*?/; # correspond avec "a"
"aaaabbbb"=~ /a+?b+?/; # correspond avec
                       # "aaaab"
```

Quantifieurs possessifs

```
x*? x+? x*?? x{n}? x{n,m}?
```

Pour répondre à des cas assez spécifiques, il existe encore une autre variation des quantifieurs qui fonctionnent de manière semblable aux quantifieurs avides mais n'effectuent jamais de retour arrière dans la recherche de correspondance. On parle de quantifieurs possessifs, ou encore de groupement atomique :

- `*+` zéro ou plusieurs fois, et ne rend jamais,
- `++` une ou plusieurs fois, et ne rend jamais,
- `?+` zéro ou une fois, et ne rend jamais,
- `{n}+` exactement *n* fois, et ne rend jamais,
- `{n,}+` au moins *n* fois, et ne rend jamais,
- `{n,m}+` entre *n* et *m* fois, et ne rend jamais.

Voici un petit exemple qui permet de comprendre les différences entre les différents types de quantifieurs :

```
# avide : /a+/ correspond avec "aaa"
"aaaa" =~ /a+a/;

# non avide : /a+?/ correspond avec "a"
"aaaa" =~ /a+?a/;

# possessif : /a++/ ne correspond pas
"aaaa" =~ /a++a/;
```

`/a++/` ne correspond pas car le motif essaye d'abord de réaliser une correspondance avec les quatre caractères `"a"`, mais le dernier est nécessaire pour correspondre avec le second caractère atome `"a"` du motif. D'où échec du motif, et comme il s'agit d'un quantifieur possessif, il n'y a pas de tentative de retour arrière, comme le fait le quantifieur avide qui revient d'une position en arrière, essaye de correspondre avec trois caractères `a` et réussit.

Capturer du texte avec les groupes capturants

```
(...)
```

Les motifs permettent de rechercher des données au sein d'un texte. C'est un bon début, mais en l'état, difficile d'en faire quelque chose de véritablement utile... En particulier, il manque un moyen pour récupérer les données intéressantes dans une variable, voire dans plusieurs variables différentes quand le motif permet d'extraire la structure des données du texte. Les *groupes capturants* répondent à ce problème.

Leur principale fonction est de pouvoir créer des éléments complexes à partir de plusieurs atomes. Leur seconde fonction est, par défaut, de capturer le texte qui a correspondu au sous-motif du groupe. La syntaxe est très simple, il suffit de mettre un sous-motif entre parenthèses `(...)` pour créer un groupe capturant.

```
"abcdefgh" =~ /b(cd)e/;     #capture "cd"
"abcdefgh" =~ /\w(\w+)\w/; #capture "bcdefg"
```

Pour se référer au texte capturé, il suffit d'utiliser les formes `\1`, `\2`, `\3`..., nommées *références arrières* au sein même de

l'expression, et les variables `$1`, `$2`, `$3`, etc. à l'extérieur.

```
# recherche d'un mot répété
if ($string =~ /(\w+) +\1/) {
       print "mot répété : $1";
}
```

Noter que ces dernières ne sont définies que dans le bloc courant et sont évidemment réinitialisées dès l'évaluation suivante de la même expression ou d'une autre expression. La durée de vie de ces variables étant donc assez courte, il est préférable de rapidement copier leurs valeurs dans des variables à soi.

```
# exemple de lecture d'un fichier
# de configuration
while ($line = <$fh>) {
       if ($line =~ /^(\w+)\s*=\s*(.*)$/) {
            $param{$1} = $2;
       }
}
```

Quand c'est possible, il est préférable d'affecter directement des variables destination par une assignation de liste :

```
while ($line = <$fh>) {
       if (my ($name, $value) =
            $line =~ /^(\w+)\s*=\s*(.*)$/) {
            $param{$name} = $value;
       }
}
```

La numérotation des captures se fait logiquement de gauche à droite, en comptant les parenthèses ouvrantes, y compris pour les groupes imbriqués :

```
"abcdefgh" =~ /(b(cd)e).+(g)/;
print "$1 : $2 : $3";
# affiche "bcde : cd : g"
```

Comme l'utilisation des références arrières numérotées posait des problèmes de syntaxe, tant au niveau de la cohérence que de la lisibilité, une nouvelle syntaxe a été introduite avec Perl 5.10, plus souple et sans ambiguïté : `\g{N}`, où *N* peut être un nombre positif, auquel cas cela correspond au numéro habituel (absolu) de la capture, ou négatif, qui permet un référencement arrière relatif. Car c'est un des gros problème de la syntaxe classique des références arrières, elles sont absolues, ce qui complique la construction d'expressions régulières complexes à partir de morceaux séparés, si ceux-ci veulent utiliser des références arrières, puisqu'il faut alors connaître le nombre global de captures. La nouvelle syntaxe permet de récupérer le texte capturé de manière relative, permettant à chaque morceau de rester indépendant. L'exemple précédent de recherche de mot répété devient alors :

```
# recherche d'un mot répété
if ($string =~ /(\w+) +\g{-1}/) {
      print "mot répété : $1";
}
```

Mieux, il est aussi possible de nommer les captures, avec la syntaxe `(?<nom>motif)` et de s'y référer avec `\g{nom}` ou `\k<nom>`. Les syntaxes PCRE et Python sont aussi supportées pour rendre plus aisée la réutilisation d'expressions régulières venant d'horizons variés. Ceci facilite encore la modularisation sous forme de morceaux contenant des motifs spécialisés :

```
my $find_dup = qr{ (?<dup_word>\w+) \s+ \g{
    dup_word} }x;
```

Pour récupérer les valeurs correspondantes en dehors de l'expression, il faut utiliser les variables lexicales `%+` et `%-` : `$+{nom}` fournit la même valeur que `\g{nom}`, et `$-{nom}` la référence vers un tableau de toutes les captures de ce

nom. Les noms de ces hashs, certes courts mais aussi peu explicites, peuvent être remplacés par des noms plus clairs, grâce au module `Tie::Hash::NamedCapture` :

```
use Tie::Hash::NamedCapture;

tie my %last_match , "Tie::Hash::Named
   ➥Capture";
# %last_match se comporte maintenant
# comme %+

tie my %reg_buffer ,"Tie::Hash::Named
   ➥Capture", all => 1;
# %reg_buffer se comporte maintenant
# comme %-
```

Bien sûr, les captures nommées restent aussi accessibles par leur numéro.

Grouper sans capturer avec les groupes non capturants

```
(?:. . .)
```

L'utilisation de groupes capturants est très pratique, mais impose aussi un coût non négligeable étant donné qu'il faut nécessairement mettre en place une mécanique interne pour récupérer et stocker le texte qui a correspondu. C'est pourquoi il est aussi possible de créer des groupes non capturants avec la syntaxe `(?:. . .)`. Cet exemple illustre l'intérêt de pouvoir grouper sans capturer :

```
# recherche simple d'une adresse IPv4
/( (?:\d+\.){3} \d+ )/x;
```

Le groupe non capturant permet de créer un sous-motif attendant plusieurs chiffres suivi d'un point, sous-motif répété trois fois afin de correspondre aux trois premières composantes de l'adresse IP, suivi enfin de la dernière composante. Les composantes individuelles ne sont pas intéressantes ici, car c'est l'adresse IPv4 dans son entier qui est voulue, d'où le groupe capturant qui englobe le tout.

Point intéressant, il est possible de positionner des modificateurs locaux au seul groupe :

```
# désactivation locale de /x
m{
  ... # expression régulière complexe
  ... # qui ne sera pas détaillée ici

  (?-x:...) # une ligne où /x
            # n'est pas désiré

  ... # et ici, retour au mode /x
}x;
```

Il est aussi possible de désactiver un modificateur en le faisant précéder par le signe moins (-), ainsi que de grouper plusieurs modificateurs :

```
# activation locale de /s et désactivation
# de /i
m/(?s-i:...) ../i;
```

Il ne faut donc pas hésiter à utiliser les groupes non capturants, leur coût à l'exécution étant négligeable.

Définir des alternatives

```
(..|..)
(?|..|..)
```

Dernier point important de la syntaxe des expressions qui n'a pas encore été abordé, les *alternatives*, qui consistent à créer des branches de correspondance possible au sein de l'expression en les séparant avec le caractère barre verticale |. Détail qui a son importance, les branches s'étendent le plus possible, jusqu'aux bords de l'expression régulière ou du groupe qui l'englobe.

Ainsi `/^krack|clonk$/` doit se comprendre comme « chercher "krack" en début de ligne, ou "clonk" en fin de ligne ». Pour obtenir un comportement du type « chercher un mot unique sur la ligne qui est soit "krack", soit "clonk" », il faut placer l'alternative dans un groupe : `/^(?:krack|clonk)$/`

Par exemple, une expression pour commencer un début de support des arguments de la commande Linux *ip* serait :

```
/^(addr(?:ess)?¦route¦rule¦table)
   ➥+(add¦del¦list)/
```

À l'exécution, les alternatives sont essayées dans l'ordre, de gauche à droite. La première qui réussit valide l'alternative.

Un problème survient lors de l'utilisation des alternatives quand les branches elles-mêmes contiennent des captures. Comme expliqué dans la partie sur les captures, celles-ci sont numérotées en comptant la parenthèse ouvrante de chaque groupe, en partant de la gauche. Pour des cas un peu complexes, cela peut donner un résultat assez curieux :

```
/ ( a )  (?: x (y) z ¦ (p(q)r) ¦ (t) u (v) )  (z) /x
# 1            2       3 4       5     6      7
```

Suivant la branche qui aura correspondu, il faudra regarder des variables différentes : `$2` pour la première, `$3` et `$4` pour la seconde, `$5` et `$6` pour la troisième. Pas forcément très pratique.

C'est pourquoi une nouveauté a été introduite en Perl 5.10, la *remise à zéro de branche* (*branch reset*), un nouveau type de groupe non capturant qui se déclare avec la syntaxe `(?|..)`. Les captures de chaque branche de l'alternative sont alors numérotées comme s'il n'y avait qu'une seule branche :

```
# avant   ---remise à zéro de branche--   après
/ ( a )  (?¦ x(y)z ¦ (p(q)r) ¦ (t)u(v) )  (z) /x
# 1           2      2 3       2   3      4
```

L'intérêt est qu'il suffit maintenant de regarder la valeur d'une seule variable, `$2`, pour savoir quelle branche a correspondu.

Bien sûr, il est aussi possible de résoudre ce genre de problèmes avec des captures nommées à la place, mais avec l'inconvénient d'une expression plus verbeuse. C'est même probablement préférable dans de nombreux cas étant donné que des champs nommés sont plus faciles à comprendre quand il s'agit de relire le code que des champs numérotés.

Découper une chaîne avec `split`

```
split(/MOTIF/, CHAINE, LIMITE)
```

La fonction `split` permet de séparer une chaîne en plusieurs éléments, délimités par le motif fourni en premier argument. Ainsi, pour récupérer les champs des lignes du fichier */etc/passwd*, séparés par des caractères deux-points :

```
my @fields = split /:/, $line;
```

Les éventuels champs vides intermédiaires sont conservés, mais les derniers sont éliminés. Si la limite est spécifiée (sous forme d'un nombre positif), cela représente le nombre maximum d'éléments dans lesquels se découpe la chaîne :

```
# les 4 premiers champs, puis le reste
# de la ligne
my @fields = split /:/, $line, 5;
```

Plus intéressant, `split` fonctionne de manière naturelle :

```
my ($login, $password) = split /:/, $line;
```

Dans un tel cas, inutile d'indiquer la limite car `split` détecte qu'il s'agit d'une assignation à une liste de variables, et positionne automatiquement la limite au nombre de variables plus un, afin de jeter le reste non utilisé. Si le comportement contraire est souhaité, il suffit de spécifier la limite :

```
my ($login, $password, $rest) = split /:/,
   ➥$line, 3;
```

Le motif peut bien sûr faire plus d'un caractère de long :

```
# moyen très rapide de lire un fichier de
# configuration de type clé=valeur et de
# stocker ses paramètres dans un hash
chomp(my @lines = grep { !/^$¦^\s*#/ }
   ➥<$config_fh>);
my %config = map { split /\s*=\s*/, $_, 2 }
   ➥@lines;
```

Si le motif contient des captures, des champs supplémentaires seront créés pour chacune, avec la valeur en cas de correspondance, `undef` sinon.

Il est possible de fournir la chaîne `" "` (une seule espace) à la place du motif, qui se comporte comme le motif `/\s+/` sauf que les éventuels éléments vides en début de la liste résultat sont supprimés, afin d'émuler le comportement par défaut de *awk*.

Si aucune chaîne n'est fournie en argument, `split` travaillera sur `$_`. Si ni la chaîne, ni le motif ne sont fournis, donc si on appelle `split` sans aucun argument, c'est équivalent à `split(" ", $_)`.

Utiliser `Regexp::Common`

```
use Regexp::Common
```

`Regexp::Common` est un des module CPAN assistant les utilisateurs dans la manipulation d'expressions régulières.

Originellement écrit par Damian Conway, il fait office de bibliothèque d'expressions régulières répondant à des besoins courants. Son interface principale est un peu baroque et passe principalement par un hash magique `%RE`. Ainsi `$RE{num}{real}` est l'expression pour tester si un nombre est un réel, et `$RE{net}{IPv4}` celle pour tester si la chaîne contient bien une adresse IPv4. Les expressions sont implémentées sous forme de modules, ce qui permet de facilement étendre `Regexp::Common` avec ses propres expressions.

Ce qui rend `Regexp::Common` plus intéressant est que ces expressions sont paramétrables et acceptent des options. Deux sont communes à toutes, `-keep` pour réaliser des

captures et -i pour travailler de manière insensible à la casse.

```
use Regexp::Common;
if ($string =~ $RE{num}{int}{-keep}) {
       $number = $1;
       $sign   = $2;
}
```

Les autres options sont spécifiques à chaque expression. Par exemple le module number supporte une option -base pour indiquer la base numérique des nombres :

```
# nombres réels en binaire
$RE{num}{real}{-base => 2}

# nombres entier en base 5
$RE{num}{int}{-base => 5}
```

et il supporte même les nombres romains avec $RE{num}{roman}.

L'intérêt de Regexp::Common est d'une part de fournir des expressions régulières *correctes* pour des besoins courants :

```
given ($host) {
 when ("localhost"       ) { say "adresse locale"}
 when (/$RE{net}{IPv4}/  ) { say "adresse IPv4" }
 when (/$RE{net}{domain}/) { say "nom d'hôte" }
 default { say "type d'argument inconnu" }
}
```

mais aussi des expressions pour des besoins moins courants et sur lesquels il est assez facile de se tromper, comme l'extraction de chaînes encadrées par des guillemets.

Le dernier point à noter avec Regexp::Common est que s'il peut sembler trop gros à charger pour certains usages, il est parfaitement possible de sauver l'expression régulière désirée quelque part pour s'en servir de manière autonome.

Utiliser Regexp::Assemble

```
use Regexp::Assemble
```

`Regexp::Assemble` est un module écrit par David Landgren pour réaliser l'assemblage de sous-motifs afin de créer une grande expression régulière partiellement optimisée. Il l'avait écrit à l'origine pour assembler des adresses d'émetteurs de pourriel et fournir ainsi une seule grosse expression à Postfix, qui supporte les expressions régulières grâce à PCRE. Ainsi, à partir d'un grand nombre d'adresses comme celles-ci,

```
64-80-231-201.fibertel.com.ar
host-67-84-230-24.midco.net
host-89-229-2-176.torun.mm.pl
host-213-213-233-44.brutele.be
ppp-58.10.219.243.revip2.asianet.co.th
68-67-200-36.atlaga.adelphia.net
```

il est facile d'en déduire des motifs simples qui y correspondent, ainsi qu'aux adresses voisines :

```
\d+-\d+-\d+-\d+\.fibertel\.com\.ar
host-\d+-\d+-\d+-\d+\.midco\.net
host-\d+-\d+-\d+-\d+\.torun\.mm\.pl
host-\d+-\d+-\d+-\d+\.brutele\.be
ppp-\d+\.\d+\.\d+\.\d+\.revip\d+\.asianet
    ➥\.co\.th
\d+-\d+-\d+-\d+\.atlaga\.adelphia\.net
```

`Regexp::Assemble` permet de réunir tous ces motifs, mais surtout le fait de manière intelligente en réalisant des groupements de préfixes partout où c'est possible :

```
(?:
     host-\d+-\d+-\d+-\d+\.
     (?:
       torun\.mm\.pl
       |brutele\.be
       |midco\.net
     )
     |\d+-\d+-\d+-\d+\.
     (?:
       atlaga\.adelphia\.net
       |fibertel\.com\.ar
     )
     |ppp-\d+\.\d+\.\d+\.\d+\.revip\d+\.
         asianet\.co\.th
)
```

Pour prendre un exemple plus court et plus simple :

```
use Regexp::Assemble;
my $ra = Regexp::Assemble->new;
$ra->add( "cra+ck", "cru+nch", "clu+n+ck" );
print $ra->re;
# affiche "(?-xism:c(?:r(?:u+nch|a+ck)|
   ➥lu+n+ck))"
```

Il faut noter que `Regexp::Assemble` ne peut véritablement assembler que des motifs comprenant des atomes simples, donc sans groupes internes. Mais même ainsi, il permet des gains sensibles de performance, et propose aussi de très intéressantes options pour par exemple suivre les branches qui établissent les correspondances, ce qui constitue une outil de déboggage très utile.

Utiliser Text::Match::FastAlternatives

```
use Text::Match::FastAlternatives
```

`Text::Match::FastAlternatives` est un module écrit par Aaron Crane alors qu'il travaillait pour le site The Register, où il maintenait un outil d'analyse de logs de plusieurs dizaines de gigaoctets, d'où des besoins de performances assez importants. Ce module est fondamentalement un moyen d'avoir l'équivalent d'un idiome assez courant dans ce cas, qui est de créer une liste d'alternatives ainsi :

```
my $re = join "¦", @words;
```

C'est facile à écrire, mais peu performant, et de plus cela n'utilise pas vraiment les expressions régulières. `Text::Match::FastAlternatives` fournit un moyen de chercher la présence d'une sous-chaîne dans un texte, et ce bien plus efficacement puisqu'il est environ 90 fois plus rapide que l'expression régulière équivalente.

Utiliser YAPE::Regex::Explain

```
use YAPE::Regex::Explain
```

`YAPE::Regex::Explain` peut s'avérer utile pour qui a encore un peu de mal avec la syntaxe des expressions régulières, car ce module permet d'expliquer une expression en termes clairs, malheureusement seulement en anglais. Ainsi le code suivant :

```
use YAPE::Regex::Explain;

my $re = qr/^(\w+)\s*=\s*(.*)$/;
my $ex = YAPE::Regex::Explain->new($re);
print $ex->explain;
```

affiche cette explication, assez verbeuse :

```
The regular expression:

(?-imsx:^(\w+)\s*=\s*(.*)$)

matches as follows:

NODE        EXPLANATION
----------------------------------------------------------
(?-imsx:    group, but do not capture (case-sensitive)
            (with ^ and $ matching normally) (with . not
            matching \n) (matching whitespace and #
            normally):
----------------------------------------------------------
  ^           the beginning of the string
----------------------------------------------------------
  (           group and capture to \1:
----------------------------------------------------------
    \w+         word characters (a-z, A-Z, 0-9, _) (1 or
                more times (matching the most amount
                possible))
----------------------------------------------------------
  )           end of \1
----------------------------------------------------------
  \s*         whitespace (\n, \r, \t, \f, and " ") (0 or
              more times (matching the most amount
              possible))
----------------------------------------------------------
  =           '='
----------------------------------------------------------
  \s*         whitespace (\n, \r, \t, \f, and " ") (0 or
              more times (matching the most amount
              possible))
----------------------------------------------------------
  (           group and capture to \2:
----------------------------------------------------------
    .*          any character except \n (0 or more times
                (matching the most amount possible))
----------------------------------------------------------
  )           end of \2
----------------------------------------------------------
```

```
  $           before an optional \n, and the end of the
              string
-----------------------------------------------------------
)           end of grouping
-----------------------------------------------------------
```

Il ne gère par ailleurs qu'une partie réduite de la syntaxe, et les ajouts récents comme les quantifieurs possessifs ne sont pas pris en compte.

Autres références

Seules les bases des expressions régulières ont été abordées ici. Si celles-ci sont relativement simples à appréhender, elles constituent pourtant un domaine véritablement complexe à maîtriser dans ses détails. Le lecteur désireux de mieux approfondir ses connaissances pourra se référer à la page du manuel Perl consacrée aux expressions régulières, *perlre*, lisible en ligne à l'adresse http://perldoc.perl.org/perlre.html, en anglais et un peu aride, mais à jour avec les dernières nouveautés.

Par ailleurs, le livre *Mastering Regular Expression, 2nd Ed* de Jeffrey Friedl (O'Reilly 2002, ISBN 0-596-00289-0) reste une référence en la matière, abordant l'ensemble du domaine lié aux expressions régulières, et décrivant les différentes implémentations, celle de Perl bien sûr mais aussi celle de PCRE, utilisée par de très nombreux autres langages et programmes, ou encore celle de Java.

Dans la même collection que le présent ouvrage, existe également le *Guide de Survie – Expressions régulières* de Bernard Desgraupes (Pearson, ISBN 978-2-7440-2253-1).

6

Concepts objet en Perl

Perl propose un grand nombre de possibilités pour écrire du code orienté objet. De la syntaxe de base faisant partie intégrante du langage aux nombreux modules plus évolués disponibles sur CPAN, il n'était pas forcément facile de savoir quelle voie suivre. . . Survint alors Perl 6, qui a commencé à poser les bases de la programmation objet que supporterait le langage, en ne prenant que le meilleur des différents langages de programmation existants. Ce nouvel ensemble forme un tout cohérent, qui a été porté sous forme de module Perl 5 : `Moose` était né. Il s'est depuis imposé comme *la* manière moderne de programmer en objet avec Perl.

Avant de plonger dans `Moose`, il est toutefois bon de poser quelques principes sur le concept d'objet en Perl. En effet, `Moose` s'appuie sur les mécanismes de base présents dans le langage pour les enrichir.

Ce chapitre mentionnera uniquement les concepts et syntaxes de base. Il ne parlera pas des notions d'objet bas niveau Perl qui ont été supplantés par `Moose`.

Créer un objet

```
Package->new( . . .)
```

Les classes fournissent un constructeur, souvent appelé `new()`[1]. Un constructeur est simplement une méthode renvoyant une variable liée – donc un objet :

```
use Personne;
my $obj = Personne->new;
```

Dans cet exemple, `$obj` est un objet de classe `Personne`. Bien sûr, il ne faut pas oublier de charger la classe avant de l'utiliser.

Qu'est-ce qu'un objet ?

En Perl, un objet est tout simplement une variable scalaire contenant une référence, avec une particularité : cette référence sait qu'elle est liée[2] à un *package* donné.

Connaître la classe d'un objet

```
ref( . . .)
```

La fonction `ref()` utilisée avec un objet permet de récupérer le package lié à cet objet :

1. Voir page 135.
2. La liaison avec le package se fait en interne avec la fonction `bless`, qui ne sera pas abordée ici.

```
my $obj = Personne->new;
say ref($obj);       # affiche "Personne"
```

La fonction `ref()` renverra dans ce cas la chaîne `Personne`.

Appeler une méthode

```
->
```

Les méthodes sont appelées en utilisant l'opérateur `->` sur l'objet, suivi de l'appel de la méthode avec ses paramètres :

```
$obj->saluer( 'matin' );
```

Classes et méthodes

Comme un objet est un scalaire lié à un package, une classe est tout simplement ledit package associé. C'est un package standard.

Les fonctions définies dans ce package deviennent des méthodes.

Définir une méthode

```
sub { }
```

Une méthode étant simplement une fonction du package, elle se définit avec le mot-clé `sub`, déjà vu dans le chapitre « Éléments du langage » page 26.

La méthode va recevoir l'objet en premier paramètre. Les paramètres passés lors de l'appel de la méthode viennent après.

```
sub saluer {
      # le premier paramètre est l'objet
      my $self = shift;
      # les paramètres d'appel viennent
          après
      say "bon @_";
}
```

Dans le cas de méthodes statiques (appelées sur la classe au lieu d'un objet), le premier paramètre est le nom de la classe elle-même :

```
package Personne;
  sub static_method {
      my $class = shift;
      say $class;
  }

Personne->static_method;
# affiche "Personne"
```

Certaines méthodes peuvent faire sens en tant que méthode statique ou méthode objet. Dans ce cas, il faudra différencier suivant le type du premier paramètre : un objet sera une référence, tandis que le nom de la classe sera juste une chaîne.

```
sub method {
      my $self = shift;
      if ( ref $self ) {
          # le paramètre est une réf. :
              méthode objet
          ...

      } else {
```

```
            # méthode statique
            ...
        }
}
```

Définir un constructeur

Une méthode spéciale du package renvoie une référence sur un nouvel objet. Cette méthode est le constructeur de la classe. Par convention, elle est souvent appelée `new()`. Cependant, cette méthode doit être écrite par le programmeur : pour Perl, cette méthode n'a rien de spécial. Il est d'ailleurs possible de créer plusieurs constructeurs, renvoyant chacun un objet : pour Perl, cela reste des méthodes comme les autres.

Classes et attributs

Un objet Perl est souvent une référence de hash liée à un package. De ce fait, les attributs sont souvent un couple clé/valeur de cette référence.

```
sub method {
 my $self = shift;
 say $self->{attribut};
}
```

Cependant, Perl est très souple et permet d'utiliser n'importe quelle référence comme objet. `Moose` permet de s'affranchir de ces détails d'implémentation, pour se concentrer sur le paradigme de la programmation orientée objet.

7

Moose

Perl fournit donc toutes les briques de base pour la programmation orientée objet : constructeur, attributs, méthodes, héritage, etc. Cependant, ces briques sont vraiment bas niveau. Un programmeur souhaitant utiliser un paradigme objet doit soit plonger dans ces détails, soit mixer un ensemble de modules objet qui fournissent chacun une surcouche au système objet de Perl... Surcouches qui ne sont pas toujours compatibles entre elles, encore moins conçues comme un tout cohérent.

`Moose` a donc été écrit pour devenir ce *tout cohérent* qui manquait au monde objet de Perl. Il fournit un ensemble de mots-clés et de concepts, cachant toute la complexité des détails d'implémentation, en laissant les auteurs se concentrer sur le problème à résoudre.

Déclarer une classe

Une classe `Moose` est un package, qui charge le module `Moose`.

```
package Personne;
use Moose;
# maintenant c'est une classe Moose !
1;
```

Le simple fait de charger le module `Moose` change l'héritage de la classe : elle devient une classe dérivée de `Moose::Object`. Celle-ci fournit entre autre un constructeur `new()`.

`Moose` va aussi activer automatiquement le mode `strict` et le mode `warnings`, pour encourager l'écriture de code propre.

Enfin, parmi les autres changements visibles, le chargement de `Moose` va importer un certain nombre de fonctions dans le package courant, qui sont détaillées dans ce chapitre.

Astuce

L'utilisation de `MooseX::Singleton` au lieu de `Moose` dans une classe change le constructeur afin qu'il renvoie tout le temps la même instance – un singleton, autrement dit.

Déclarer un attribut

has

`Moose` permet bien sûr de déclarer les attributs d'une classe facilement avec le mot-clé `has`[1] :

```
package Personne;
use Moose;
has nom => ( is=>'rw' );
```

Info

Les objets possèdent la plupart du temps des *attributs*. Un attribut est une variable privée que tous les membres de la classe

1. *Has* veut dire « possède » en anglais.

possèdent. Par exemple, tous les objets d'une classe `Personne` auront un nom et une date de naissance.

Les options passées à la fonction `has` définissent les propriétés de l'attribut. `Moose` propose un panel complet d'options pour personnaliser l'attribut selon son but. Le seul paramètre obligatoire est `is`, qui permet de définir si l'attribut sera accessible en lecture seule (valeur `ro` du paramètre) ou pourra être modifié durant la vie de l'objet (`rw`).

Ainsi, dans l'exemple précédent, tout objet `Personne` aura un attribut nom optionnel qui pourra être modifié au cours de la vie de l'objet.

Astuce

Si une classe déclare plusieurs attributs identiques (mis à part le nom, bien sûr), il est possible de les déclarer tous d'un coup :

```
package Couleur;
use Moose;
has [ 'rouge', 'vert', 'bleu' ]
   => ( is=>'rw', isa=>'Int' );
```

Il est possible de spécifier la valeur d'un attribut lors de l'instanciation d'un objet, en passant le nom de l'attribut et sa valeur comme paramètres lors de l'appel du constructeur :

```
my $p = Personne->new(nom=>'Jerome Quelin');
```

Et si la classe `Personne` avait défini deux attributs `nom` et `prenom` :

```
my $p = Personne->new( prenom=>'Jerome',
                       nom=>'Quelin'
                     );
```

L'ordre de passage des paramètres n'a pas d'importance.

Accéder aux objets

Dès qu'un attribut est déclaré avec `has`, `Moose` va générer un *accesseur* pour cet attribut, du nom de cet attribut :

```
package Personne;
use Moose;
has nom => ( is=>'rw' );

my $p = Personne->new( nom=>'Jerome' );
say $p->nom;
    # affiche "Jerome"
$p->nom( 'Jerome Quelin' );
say $p->nom;
    # affiche "Jerome Quelin"
```

La méthode est donc un *mutateur*, agissant comme un accesseur (*getter*) ou un modificateur (*setter*) suivant si des paramètres lui sont passés. Si l'objet est déclaré en lecture seule, la méthode ne pourra pas être utilisée pour modifier la valeur de l'attribut :

```
package Personne;
use Moose;
has nom => ( is=>'ro' );

my $p = Personne->new;
$p->nom( 'Jerome Quelin' );      # invalide
```

Tenter d'exécuter ce programme conduira à l'erreur :

```
Cannot assign a value to a read-only
accessor at ./test line 13
```

Modifier le nom des accesseurs

```
reader
writer
```

`Moose` permet de modifier le nom des accesseurs à l'aide des paramètres `reader` et `writer`, qui changent respectivement le nom de l'accesseur et celui du modificateur.

Changer le nom d'un accesseur peut se révéler utile pour changer l'API de la classe. Par exemple, pour rendre un attribut publiquement lisible, mais uniquement modifiable de façon privée :

```
package Personne;
use Moose;
has gps_coords => ( is => 'rw',
                    writer => '
                        _set_gps_coords'
                  );
```

Info

La convention Perl consiste à préfixer par un _ les attributs et méthodes privés. Cela n'empêche pas les classes extérieures d'utiliser ces attributs et méthodes, mais c'est à leur risque !

Cela est utile dans le cas d'attributs calculés, mis à jour par des méthodes de la classe. Ainsi, la classe `Personne` ci-dessus peut définir une méthode `move()` qui va modifier la localisation de la personne à chaque appel, permettant de masquer la complexité du changement de lieu tout en permettant à l'utilisateur de la classe de savoir où se trouve cette personne *via* `$p->gps_coords`,

Certains préfèrent un style de programmation avec des méthodes séparées pour les accesseurs et les modificateurs, ce qui peut aussi se faire avec le changement de nom des accesseurs :

```
package Personne;
use Moose;
has nom => (
```

```
      is => 'rw',
      reader => 'get_nom',
      writer => 'set_nom',
);

my $p = Personne->new;
$p->set_nom( 'Jerome Quelin' );
say $p->get_nom;
    # affiche "Jerome Quelin"
```

Bien sûr, cela devient vite fastidieux s'il faut le faire pour chaque attribut... Le module `Moose::FollowPBP`[2] permet d'automatiser cette pratique :

```
package Personne;
use Moose;
use MooseX::FollowPBP;
has nom => ( is => 'rw' );
my $p = Personne->new;
$p->set_nom( 'Jerome Quelin' );
say $p->get_nom;
    # affiche "Jerome Quelin"
```

Une autre extension de `Moose`, `MooseX::SemiAffordance-Accessor`, permet d'avoir les accesseurs du nom de l'attribut et les modificateurs précédés de `set_` :

```
package Personne;
use Moose;
use MooseX::SemiAffordanceAccessor;
has nom => ( is => 'rw' );
my $p = Personne->new;
$p->set_nom( 'Jerome Quelin' );
say $p->nom;
# affiche "Jerome Quelin"
```

2. PBP comme *Perl Best Practices*, du nom du livre de Damian Conway, publié aux Éditions O'Reilly.

Méthodes de prédicat et de suppression

```
predicate clearer
```

`Moose` permet de faire la distinction entre une valeur fausse (au sens booléen du terme) et une valeur non définie.

Cette distinction se fait grâce à la définition de méthodes de prédicat, qui vont renvoyer vrai si une valeur a bien été spécifiée pour cet attribut – et ce, même si la valeur spécifiée est fausse.

Définir une méthode de suppression permet d'effacer la valeur d'un attribut. Cette action est différente de celle qui consiste à réinitialiser l'attribut à `undef` : elle change aussi la valeur de retour de la méthode prédicat.

```
package Personne;
use Moose;
has naissance => (
      is => 'rw',
      # vérification de l'existence
      predicate => 'has_naissance',
      # suppression de la valeur
      clearer   => 'clear_naissance'
);

my $p = Personne->new;
$p->naissance;                  # undef
$p->has_naissance;              # faux

$p->naissance(undef);
$p->naissance;                  # undef
$p->has_naissance;              # vrai

$p->clear_naissance;
$p->naissance;                  # undef
$p->has_naissance;              # faux
```

```
$p->naissance( '1976/10/18' );
$p->naissance;                  # "1976/10/18"
$p->has_naissance;              # vrai

my $epoch = Personne->new( naissance =>
   ➥'1970/01/01' );
$epoch->has_naissance;          # vrai
```

Ce besoin n'est cependant pas très répandu, `Moose` ne crée donc pas ces méthodes automatiquement. Il faudra spécifier les paramètres `predicate` et `clearer` lors de l'appel de `has` pour que `Moose` génère ces méthodes.

Rendre un attribut obligatoire

`required`

Par défaut, les attributs sont optionnels : ils ne sont pas obligatoirement fournis lors de l'appel au constructeur.

`Moose` peut cependant rendre un attribut obligatoire grâce au paramètre `required` lors de l'appel de `has` :

```
package Personne;
use Moose;
has nom => ( is => 'ro', required => 1 );

my $p1 = Personne->new;          # erreur
my $p2 = Personne->new( nom=>'Jerome
   ➥Quelin' );                  # valide
```

Attention

Passer une valeur `undef` lors du constructeur est tout à fait possible. De même, spécifier une méthode de suppression (*cf.* ci-dessus) pour cet attribut est aussi valide, même si cela n'a guère de sens…

Vérifier le type d'un attribut

isa

Heureusement, `Moose` possède un solide système de typage (voir Chapitre 8) permettant de spécifier les valeurs qu'un attribut peut prendre. Le type d'un attribut se définit *via* le paramètre `isa`[3].

```
package Personne;
use Moose;
has nom => ( is => 'ro', required => 1,
   ➥isa => 'Str' );
```

Ainsi, il n'est plus possible de passer une valeur `undef` pour le nom lors de l'appel du constructeur : `Moose` lancera une exception qui arrêtera le programme si elle n'est pas traitée.

Donner une valeur par défaut

default

Il est courant de proposer des valeurs par défaut aux attributs, ce qui est supporté par `Moose` :

```
package Personne;
use Moose;
has pays => (
      is => 'rw', isa => 'Str', required =>
          1,
      default => 'France'
);
```

3. Se lisant *is a*, ou *est un*.

```
# valide, malgré le required
my $p = Personne->new;

# affiche "France"
say $p->pays;
```

Si l'appel au constructeur ne contient pas d'attribut `pays`, celui-ci sera initialisé à `France`. Si l'attribut avait un prédicat, celui-ci renverra vrai (c'est-à-dire que l'attribut a une valeur). Ainsi, il n'est plus nécessaire de spécifier l'argument `required` pour les attributs proposant une valeur par défaut.

Il est aussi possible de fournir une référence à une fonction comme valeur par défaut. Dans ce cas, elle sera appelée par `Moose` comme une méthode sur l'objet nouvellement créé :

```
package Personne;
use Moose;
has numero_fetiche => (
    is => 'rw', isa => 'Str',
    default => sub {
        my @nbs = (1..9);
        return $nbs[ rand @nbs ];
    },
);
```

Utiliser une fonction de rappel est aussi nécessaire pour fournir une valeur par défaut aux références. En effet, sans cette précaution, Perl n'initialiserait qu'une seule référence et l'utiliserait comme valeur par défaut pour tous les objets... Et ainsi tous les attributs des objets pointeraient vers la même valeur ! `Moose` détecte donc ce problème et interdit l'utilisation d'une référence comme valeur par défaut, en forçant l'utilisation d'une fonction de rappel :

```
package Personne;
use Moose;
has liste_courses => (
      is => 'rw', isa => 'ArrayRef',
      # création d'une liste unique
      default => sub { [] },
);
```

Ce n'est pas très élégant, mais c'est dû à la sémantique Perl elle-même.

Construire un attribut

builder

Plutôt qu'utiliser une fonction de rappel, `Moose` permet d'utiliser une méthode *constructeur d'attribut*, grâce au paramètre `builder` :

```
package Personne;
use Moose;
has numero_fetiche => (
      is => 'rw', isa => 'Str',
      builder => '_build_numero_fetiche',
);
sub _build_numero_fetiche {
      my @nbs = (1..9);
      return $nbs[ rand @nbs ];
}
```

Outre le gain évident de lisibilité dû à la différentiation des déclarations et du code, cette notation a deux grands avantages.

Tout d'abord, elle permet facilement aux classes filles de modifier cette valeur par défaut, en surchargeant cette méthode (voir la section sur l'héritage page 154) :

```
package Enfant;
use Moose;
extends 'Personne';
sub _build_numero_fetiche { 7 }
```

Ensuite, cette méthode permet d'utiliser des rôles. Un rôle peut donc fournir un attribut, mais requérir que la classe fournisse le constructeur (voir page 156).

Rendre un attribut paresseux

lazy

Une classe peut définir un grand nombre d'attributs, chacun avec son constructeur. Cependant, si certains attributs sont rarement utilisés, cela implique que de la mémoire sera mobilisée pour eux aussi, jusqu'à la destruction de l'objet. Sans compter le temps processeur utilisé pour générer ces attributs – qui, même s'il est dérisoire, peut s'additionner dans le cas de millions d'objets avec des dizaines d'attributs.

`Moose` permet donc de définir des attributs « paresseux », dont le constructeur ne sera appelé que lorsque l'accesseur sera appelé. Cela se fait grâce à la propriété `lazy` :

```
package Personne;
use Moose;
has no_securite_sociale => (
    is      => 'rw',
    lazy    => 1,
    builder => '_build_no_securite_sociale',
);
```

Les attributs paresseux ont aussi une autre utilité : si un attribut a une valeur par défaut dépendant d'autres attributs, cet attribut *doit* être paresseux. En effet, `Moose` ne garantit pas l'ordre de génération des valeurs par défaut : il n'est donc pas possible de compter sur la valeur d'un autre attribut. . . sauf à utiliser un attribut paresseux :

```
package Personne;
use Moose;
has pays => ( is=>'rw', isa=>'Str',
   ➥default=>'France' );
has boisson => (
    is=>'rw', isa=>'Str',
    lazy => 1, # permet de différer l'appel
    builder => '_build_boisson',
);
sub _build_boisson {
    my $self = shift;
    return $self->pays eq 'France' ? 'vin' :
   ➥ 'bière';
}

my $p = Personne->new;
say $p->boisson;  # affiche 'vin'
```

Astuce

Il est recommandé de rendre paresseux tous les attributs ayant un constructeur (ou une valeur par défaut générée de manière un tant soit peu complexe). Cela est d'ailleurs facilité par `Moose` avec le paramètre `lazy_build` :

```
has boisson => ( is=>'rw', lazy_build=>1 );
# équivalent à :
#   has boisson => (
#       is        => 'rw',
#       lazy      => 1,
#       builder   => '_build_boisson',
#       clearer   => 'clear_boisson',
#       predicate => 'has_boisson',
#   );
```

Pour les attributs dits privés[4], les méthodes générées le seront aussi :

```
has _boisson => ( is=>'rw', lazy_build=>1 );
# équivalent à :
#   has boisson => (
#       is         => 'rw',
#       lazy       => 1,
#       builder    => '_build__boisson',
#       clearer    => '_clear_boisson',
#       predicate  => '_has_boisson',
#   );
```

Notez le double __ dans le nom du *builder*.

Spécifier un déclencheur

trigger

Les déclencheurs (ou *triggers* en anglais) comptent parmi les fonctionnalités avancées de `Moose`. Ils permettent de spécifier une fonction de rappel qui sera appelée lorsqu'un attribut est modifié :

```
package Personne;
use Moose;
has pays => (
      is      => 'rw',
      trigger => \&_set_pays,
);

sub _set_pays {
      my ($self, $new, $old) = @_;
      my $msg = "déménagement ";
      $msg .= "de $old " if @_ > 2;
      $msg .= "vers $new";
```

4. Commençant par un _.

```
        say $msg;
}
my $p = Personne->new;
$p->pays( 'France' );
# affiche "déménagement vers France"
$p->pays( 'USA' );
# affiche "déménagement de France vers USA"
```

Le déclencheur est appelé comme une méthode de l'objet courant. Il reçoit aussi la nouvelle valeur et l'ancienne (si celle-ci existait). Cela permet de différencier les cas où l'ancienne valeur n'existait pas de ceux où elle valait `undef`.

Déréférencer un attribut

auto_deref

Les attributs de type référence peuvent utiliser la propriété `auto_deref` pour que `Moose` déréférence automatiquement la valeur lors d'un appel à l'accesseur :

```
package Personne;
use Moose;
has liste_courses => (
        is => 'rw', isa => 'ArrayRef',
        default => sub { [] },
);
my $p = Personne->new;
my @liste = $p->liste_courses;
# renvoie une vraie liste !
```

Ce mécanisme est pratique, car les attributs de type liste ou hash sont obligatoirement des références. Le paramètre `auto_deref` permet donc de s'affranchir des références pour manipuler directement le type Perl.

Affaiblir un attribut référence

```
weak_ref
```

Il est possible de marquer un attribut de type référence comme étant une référence dite « faible ». Ces références ne vont pas augmenter le compteur de références de la variable[5] et permettront donc à Perl de libérer la mémoire dès que possible :

```
package Personne;
use Moose;
has parent => ( is=>'rw', isa=>'Personne',
                weak_ref=>1
                );

my $child = Personne->new;
{
    my $parent = Personne->new;
    $child->parent( $parent );
}   # fin de portée : $parent est libéré !
```

À la sortie de la portée ci-dessus, `$child->parent` est vide, car `$parent` a été réclamé par Perl. En effet, la référence stockée dans `$child->parent` était « faible » et ne suffisait pas pour maintenir `$parent` en vie. Ce mécanisme est très commode pour créer des structure de données circulaires, telles que graphes ou autres listes chaînées.

Chaîner les attributs

```
traits => [ 'Chained' ]
```

5. Perl n'utilise pas de ramasse-miettes en interne, mais des compteurs de références pour savoir si une valeur est toujours utilisée.

L'extension `MooseX::ChainedAccessor` permet de remanier les modificateurs pour qu'ils renvoient l'objet. Cela permet de chaîner les attributs :

```
package Test;
use Moose;
has debug => (
      is      => 'rw',
      isa     => 'Bool',
      traits => [ 'Chained' ],
);
sub run { ... }

my $test = Test->new;
$test->debug(1)->run();
```

C'est particulièrement intéressant pour les objets ayant un grand nombre d'attributs qui seront modifiés durant la vie de l'objet.

Simplifier les déclarations d'attributs

```
use MooseX::Has::Sugar
```

L'extension `MooseX::Has::Sugar` exporte un certain nombre de fonctions permettant de déclarer des attributs plus facilement. La déclaration suivante :

```
has attr => ( is=>'rw', required=>1,
              weak_ref=>1, auto_deref=>1
            );
```

est équivalente à :

```
use MooseX::Has::Sugar;
has attr => ( rw, required, weak_ref,
   ➥auto_deref );
```

Les classes ayant un nombre d'attributs important en bénéficieront donc grandement.

Étendre une classe parente

extends

L'héritage se fait grâce au mot-clé `extends` :

```
package Personne;
use Moose;
has nom    => ( is=>'ro', isa=>'Str',
   ➥required=>1 );
has prenom => ( is=>'ro', isa=>'Str',
   ➥required=>1 );
1;

package User;
use Moose;
extends 'Personne';
has login => ( is=>'ro', isa=>'Str',
   ➥required=>1 );
1;
```

Info

Une des fonctionnalités importantes de la programmation objet est *l'héritage*. L'héritage permet d'écrire une classe en étendant une classe de base, en ajoutant ce qui manque à la classe parente, voire en réécrivant certains comportements qui changent.

Un objet de la classe `User` aura donc un nom, un prénom et un login.

Il est possible de faire appel à `Moose` pour hériter d'une classe n'utilisant pas `Moose`. Dans ce cas, la classe va hériter

d'un constructeur qui ne connait pas le système `Moose` : un grand nombre de raccourcis liés à `Moose` ne fonctionneront plus, il faudra alors (entre autre) initialiser les attributs à la main.

En cas d'héritage multiple, il faut lister les classes parentes dans le même appel à `extends`, sinon chaque appel à `extends` réinitialiserait l'arbre d'héritage de la classe.

```
package Carre;
use Moose;
extends 'Rectangle', 'Losange';
# héritage multiple
1;
```

Surcharger une méthode

Surcharger une méthode dans une classe fille se fait de la plus simple des manières, en écrivant une nouvelle fonction du même nom dans la sous-classe :

```
package Animal;
use Moose;
sub crie { say '...' }

package Chien;
use Moose;
extends 'Animal';
sub crie { say 'ouah!'; }

my $c = Chien->new;
$c->crie;   # affiche "ouah!"
```

Il est possible de modifier plus finement le comportement des méthodes héritées avec les modificateurs de méthode (voir page 171).

Modifier des attributs hérités

```
has '+attribut' => . . .
```

Par défaut, une classe hérite de tous les attributs de ses parents. Il est cependant possible de modifier certains aspects de ces attributs, tels que les valeurs par défaut, la paresse de l'attribut, son type, etc.

Pour cela, il faut redéfinir l'attribut avec la fonction `has` en le préfixant avec un `+`, avant de modifier les paramètres à changer :

```
package Francais;
use Moose;
extends 'Personne';
has '+nom' => ( default => 'Dupont' );
```

Info

Il est cependant plus facile d'utiliser un constructeur pour l'attribut. De plus, changer les propriétés d'un attribut peut facilement mener à des bugs subtils. Cette pratique n'est donc pas à utiliser à la légère.

Créer un rôle

```
use Moose::Role
```

Rôles

Les rôles sont une alternative à la hiérarchie habituelle de la programmation orientée objet.

Un rôle encapsule un ensemble de comportements qui peuvent être partagés entre différentes classes, potentiellement sans relation entre elles. Un rôle n'est pas une classe, mais un rôle est « consommé » par une classe. En pratique, les attributs et méthodes du rôle apparaîssent dans la classe comme si elle les avait définis elle-même. Une classe héritant de la classe ayant appliqué les rôles héritera aussi de ces méthodes et attributs.

Les rôles sont à peu près l'équivalent des traits de SmallTalk, des interfaces Java, ou des mixins de Ruby.

En plus de définir des méthodes et attributs, un rôle peut requérir que la classe définisse certaines méthodes.

Créer un rôle se fait de la même manière que créer une classe, mais en utilisant cette fois `Moose::Role` :

```
package AnimalDeCompagnie;
use Moose::Role;

has nom => ( is=>'rw', isa=>'Str' );
sub caresser { say 'purrrr'; }
```

Le rôle `AnimalDeCompagnie` est un rôle définissant un attribut `nom`, et une méthode `caresser()`.

Attention

Il n'est pas possible d'instancier un rôle.

Consommer un rôle

`with`

Appliquer un rôle à une classe se fait avec la fonction `with` :

```
package Chien;
use Moose;
extends 'Mammifere';
with 'AnimalDeCompagnie';
```

La classe `Chien` a donc maintenant un attribut nom, et une méthode `caresser()`.

Il est possible de composer plusieurs rôles au sein d'une même classe :

```
package Chien;
use Moose;
with 'AnimalDeCompagnie', 'GuideAveugle';
```

Nous voyons bien que les rôles fournissent une alternative intéressante à l'héritage classique, car les rôles `AnimalDeCompagnie` et `GuideAveugle` sont plus des fonctions que des propriétés intrinsèques de la classe.

Requérir une méthode

requires

Un rôle peut forcer la classe qui le compose à fournir une méthode. L'exemple ci-dessus fournissait la méthode `caresser()`, alors qu'il est plus judicieux de laisser la classe fournir cette méthode.

```
package AnimalDeCompagnie;
use Moose::Role;

requires 'caresser';
has nom => ( is=>'rw', isa=>'Str' );
```

Les classes `Chat` et `Chien` pourront maintenant fournir un comportement adapté lors de l'appel à `caresser()`. Une exception sera levée si la classe composant le rôle ne fournit pas cette méthode.

Un accesseur de la classe composant le rôle peut très bien acquitter le prérequis spécifié par le rôle. Dans ce cas, l'attribut générant l'accesseur doit être défini avant la composition du rôle.

Le rôle peut aussi ajouter des modificateurs de méthodes (voir page 171) pour s'assurer d'un comportement – ceci est une combinaison très puissante. Le rôle `AnimalDeCompagnie` pourrait alors augmenter un attribut `contentement` après un appel à `caresser()`.

Construire et détruire des objets

Une classe utilisant `Moose` devient une classe dérivée de `Moose::Object`, qui fournit un constructeur appelé `new`. Elle ne doit donc pas redéfinir ce constructeur, qui cache une bonne partie de la magie de `Moose`.

Pour modifier son comportement, `Moose` propose un ensemble de points d'ancrage. Si ces méthodes sont définies dans la classe, elles seront appelées lors de la construction de l'objet.

Modifier les paramètres du constructeur

```
around BUILDARGS => sub { . . .}
```

`Moose` utilise un mécanisme clé/valeur pour le passage d'arguments. Il peut être pratique de changer ce mécanisme, par exemple lorsque la classe n'a qu'un seul attribut.

Cela se fait en modifiant la méthode BUILDARGS (voir page 171) :

```
package Personne;
use Moose;
has nom => ( is=>'ro', isa=>'Str' );
around BUILDARGS => sub {
  my $orig  = shift;
  my $class = shift;

  return $class->$orig(nom=>@_) if @_ == 1;
  return $class->$orig(@_);
};

my $p = Personne->new( 'Jerome Quelin' );
```

Interagir avec un objet nouvellement créé

```
sub BUILD { . . .}
```

Si une classe définit une méthode BUILD, celle-ci sera appelée *après* qu'un objet a été créé. Cela peut permettre de faire un certain nombre de validations sur l'objet lui-même, initier un traitement sur l'objet... ou tout simplement enregistrer la création d'un nouvel objet :

```
sub BUILD {
      my $self = shift;
      debug( "nouvel objet créé : $self" );
}
```

Les méthodes BUILD des classes parentes sont automatiquement appelées (si elles existent), depuis la classe parente jusqu'à la classe fille. Cet ordre permet aux classes parentes de faire les initialisations nécessaires avant que les

spécificités de la classe fille ne soient prises en compte – celles-ci dépendant en effet souvent de l'état de base des classes parentes.

Interagir lors de la destruction d'un objet

```
sub DEMOLISH { . . .}
```

De la même manière que `BUILD` est appelé après la création d'un objet, la méthode `DEMOLISH` est appelée *avant* la destruction d'un objet.

```
sub DEMOLISH {
      my $self = shift;
      debug( "objet détruit : $self" );
}
```

`Moose` appellera cette fois les méthodes `DEMOLISH` en partant de la classe la plus profonde pour remonter dans la hiérarchie, de la classe la plus spécifique à la plus générale.

Cette méthode n'est pas un destructeur : l'objet lui-même sera détruit par Perl lorsqu'il ne sera plus référencé. Cette méthode n'est donc pas nécessaire la plupart du temps : elle ne doit servir que pour des finalisations externes, par exemple en cassant la connexion à une base de données.

8

Le typage dans Moose

Les types dans `Moose` sont des objets. Ils définissent un certain nombre de contraintes et peuvent être hiérarchisés. `Moose` dispose de types de base et il est possible de créer de nouveaux types et sous-types.

Utiliser les types de base

```
Bool Undef Maybe Defined
```

Voici les principaux types reconnus par `Moose` :

- `Bool` : un booléen, acceptant soit la valeur `1` (vrai) ou toute valeur que Perl traite comme faux (faux).
- `Undef` : la valeur `undef`.
- `Maybe[type]` : `Moose` acceptera soit une valeur du `type` donné, soit `undef`. Par exemple :

```
package Personne;
has parent => ( is => 'rw', isa =>
   ➥'Maybe[Personne]' );
```

- `Defined` : toute valeur définie (non `undef`, donc). Ce type possède de nombreuses sous-catégories, hiérarchisées :
  - `Value` : une valeur. Les sous-catégories sont `Str` pour les chaînes, `Num` pour tout ce que Perl accepte comme un nombre et `Int` pour les entiers. À noter qu'un `Int` est aussi un `Num`, qui est aussi un `Str`.
  - `ClassName` ou `RoleName` : le nom d'une classe ou d'un rôle (voir page 156). Dans ce cas, le type n'acceptera que les objets de la classe (ou du rôle) donné :

```
package Personne;
use Moose;
has parent => ( is=>'rw', isa=>'
    Personne' );
```

    L'attribut `parent` n'acceptera qu'une autre `Personne` comme valeur.

  - `Ref` : une référence. Les sous-catégories sont `ScalarRef`, `ArrayRef`, `HashRef`, `CodeRef`, `FileHandle`, `RegexpRef` et `Object`. Les trois premiers types peuvent être encore restreints, pour forcer un type sur les scalaires pointés par la référence. Par exemple, pour n'accepter qu'une liste contenant des entiers :

```
package Loto;
use Moose;
has numéros_gagnants
  => ( is=>'rw', isa=>'ArrayRef[Int]'
      );
```

    Le type `Object` comporte toutes les références à un objet Perl, même si celui-ci n'est pas un objet d'une classe `Moose`.

Créer une bibliothèque de types personnalisés

```
use Moose::Util::TypeConstraints
```

Il est recommandé de regrouper les définitions des types personnalisés d'une application dans un module qui ne fera que cela. Cette bibliothèque de types utilisera le module `Moose::Util::TypeConstraints` qui a deux rôles :

- **Importation.** Il importe toutes les fonctions permettant de travailler avec les types (voir ci-dessous).
- **Exportation.** Il exporte automatiquement tous les nouveaux types créés dans le module.

`Moose` n'a qu'un seul espace de noms pour tous les types (quel que soit le module définissant les types). Si une application définit sa propre bibliothèque de types, il lui faut donc s'assurer que le nom de ses types personnalisés n'entrent pas en conflit avec les types définis par d'autres modules utilisés dans l'application. Pour cela, il est conseillé de préfixer les types personnalisés – par exemple `My::App::Types::NombrePair`.

Définir le type `NombrePair` dans le module `My::App::Types` n'est *pas* la même chose : si un autre module définit aussi un type `NombrePair`, il y aura un conflit de nom sur ce type.

Définir un sous-type

```
subtype
```

Un sous-type est défini à partir d'un type parent et d'une contrainte. Bien sûr, pour qu'une valeur soit valide, elle

doit passer les contraintes du parent (vérifiées en premier), ainsi que les contraintes additionnelles du sous-type.

Il est aussi possible de définir un message d'erreur spécifique personnalisé en cas de non-vérification des contraintes du sous-type.

```
use Moose::Util::TypeConstraints;
subtype NombrePair
      => as 'Int'
      => where { $_ % 2 == 0 }
      => message { "Le nombre $_ n'est pas
          pair." };
```

Les fonctions `as`, `where` et `message` (et d'autres encore), permettant de travailler avec les types de `Moose`, sont exportées par le module `Moose::Util::TypeConstraints`.

Définir un nouveau type

type

Il est possible de définir un type qui ne soit pas un sous-type d'un type déjà existant. Cela se fait avec le mot-clé `type` de la même manière qu'un sous-type :

```
use Moose::Util::TypeConstraints;
type UnCaractere
   => where { defined $_ && length $_ == 1
       };
```

Info

Il est cependant difficile de trouver un cas où la création d'un type ne puisse être remplacé par la création d'un sous-type, fut-il très générique (`Defined`, `Ref` ou `Object`).

L'exemple précédent aurait par exemple été plus correct en tant que sous-type de `Str`.

Définir une énumération

```
enum
```

`Moose` permet la création d'un sous-type contenant un ensemble de chaînes. Ce sous-type dépend du type de base `Str`, ajoutant une contrainte forçant la valeur à avoir l'une des valeurs listées (en tenant compte de la casse) :

```
enum LinuxDistributions
  => qw{ Fedora Ubuntu Mandriva };
```

À la différence d'autres langages (comme le C par exemple), il ne s'agit pas vraiment d'un réel type énuméré affectant une valeur entière à chaque membre de l'énumération : c'est juste un moyen rapide de générer une contrainte sur une liste de valeurs.

Définir une union de types

```
|
```

`Moose` permet à un attribut d'être de plusieurs types différents. Cela se fait avec une union de types :

```
has os => (is => 'ro', isa => 'Linux¦BSD');
```

Dans l'exemple ci-dessus, l'attribut `os` peut être soit un objet d'une classe `Linux`, soit un objet d'une class `BSD`.

Cependant, dans ces cas, il peut être préférable d'utiliser :

- Soit un rôle que les classes implémentent. Dans ce cas le type à spécifier est le nom du rôle.
- Soit du transtypage (voir ci-dessous). Dans ce cas le type à spécifier est le type vers lequel les valeurs seront transtypées.

L'union de type est donc bien souvent inutile et avantageusement remplacée par un autre mécanisme plus lisible.

Transtyper une valeur

coerce . . .from . . .via

Le transtypage consiste à convertir une valeur d'un type vers un autre. C'est une fonctionalité assez puissante, à utiliser toutefois avec précaution.

`Moose` permet de transtyper, mais il faut pour cela deux conditions :

- *Condition 1* : le type cible doit savoir comment extraire la valeur depuis le type source. Cela se fait après la définition du type, avec la fonction `coerce` :

```
subtype HexNum
      => as 'Str'
      => where { /^0x[a-f0-9]+$/i };

coerce Int
      => from 'HexNum'
      => via { hex substr $_, 2 };
```

Le type `Int` sait maintenant convertir une valeur de type `HexNum` en extrayant la valeur hexadécimale de la chaîne[1].

- *Condition 2* : l'attribut doit spécifier qu'il accepte d'être transtypé par `Moose`. Le transtypage est en effet une opération efficace mais qui peut être source de bugs difficiles à diagnostiquer et trouver. Par défaut, `Moose` refuse donc de transtyper et force l'utilisateur à se montrer explicite :

```
has i => ( is=>'ro', isa=>'Int',
   ➥coerce=>1 );
```

Le paramètre `coerce` dans la définition de l'attribut indique à `Moose` de faire les opérations de transtypage sur cet attribut.

1. Certes, Perl reconnaît déjà la notation `0x...` pour définir un entier, mais ce morceau de code a valeur d'exemple.

9

Moose et les méthodes

Une méthode reste une fonction du package, appelée suivant les mêmes conventions de Perl. `Moose` fournit cependant un certain nombre de fonctionnalités assez pratiques autour des méthodes.

Modifier des méthodes

Les *modificateurs de méthodes* sont des méthodes qui seront automatiquement appelées par `Moose` lors de l'appel de la méthode originale. Cela est bien utile, en particulier pour les méthodes héritées de classes parentes.

Les sections suivantes modifieront la classe suivante, qui servira de base :

```
package Exemple;
use Moose;

sub method { say 'dans method()'; }

my $obj = Exemple->new;
```

Le morceau de code crée donc une classe `Exemple` qui définit une méthode `method()` affichant un message. Une instance `$obj` de la classe `Exemple` est alors créée.

Intercaler un prétraitement

`before`

Il est possible d'intercaler une méthode qui sera appelée *avant* l'appel à la méthode :

```
before method => sub { say 'avant
   ➥method() 1'; };
```

L'appel à la méthode *via* `$obj->method()` affichera maintenant :

```
avant method() 1
dans method()
```

La méthode appelée avant est une méthode comme une autre, recevant donc l'objet en premier paramètre, suivi des paramètres d'appel orginaux. Sa valeur de retour est ignorée.

Ce modificateur est intéressant pour étendre le fonctionnement des méthodes auto-générées par `Moose` comme les accesseurs ou pour faire de la validation avant d'appeler la méthode :

```
before move => sub {
    my $self = shift;
    die "un véhicule est nécessaire pour
        se déplacer"
        if not $self->has_vehicle;
};
```

Définir un deuxième puis d'autres prétraitement est possible, ils seront dans ce cas appelés dans l'ordre *inverse* de définition :

```
before method => sub { say 'avant
   ➥method() 2'; };
# $obj->method() affichera :
#    avant method() 2
#    avant method() 1
#    dans method()
```

Comme le modificateur de méthode est implémenté par une fonction, il faut bien terminer la déclaration par un point-virgule.

Intercaler un post-traitement

after

De la même manière, `Moose` permet d'intercaler une méthode qui sera appelée juste *après* l'appel à cette méthode :

```
after method => sub { say 'après
   ➥method() 1'; };
```

L'affichage lors de l'appel de `$obj->method()` devient donc :

```
avant method() 2
avant method() 1
dans method()
après method() 1
```

Tout comme avec `before`, la valeur de retour de cette méthode est aussi ignorée.

Il est bien sûr ici aussi possible d'intercaler plusieurs méthodes, par symétrie, celles-ci seront alors appelées dans l'ordre de leur définition :

```
after method => sub { say 'après
   ➥method() 2'; };
# $obj->method() affichera :
#    avant method() 2
#    avant method() 1
#    dans method()
#    après method() 1
#    après method() 2
```

S'intercaler autour d'une méthode

around

Enfin, `Moose` permet de s'intercaler *autour* de l'appel d'une méthode, *via* la fonction `around`. Elle est plus puissante que les modificateurs `before` et `after`, car elle permet de modifier les arguments passés à la méthode, et même de ne pas appeler la méthode originale ! Elle permet aussi de modifier la valeur de retour.

La méthode entourant la méthode originale reçoit cette dernière en premier paramètre, puis l'objet[1] et enfin les paramètres passés à la méthode :

```
around method => sub {
  my $orig = shift;  # méthode originale
  my $self = shift;  # objet
  say 'autour de method() 1';
  $self->$orig(@_);  # appel de la méthode
                     # originale
  say 'autour de method() 1';
};
```

1. Cela est donc différent de l'ordre de passage habituel voulant que l'objet soit le premier paramètre.

Les méthodes insérées avec `around` seront appelées après les prétraitements et avant les post-traitements. De plus, la définition de plusieurs modificateurs suit la même logique que précédemment : ordre inverse de définition avant la méthode, ordre de définition après la méthode. L'affichage devient donc :

```
avant method() 2
avant method() 1
autour de method() 2
autour de method() 1
dans method()
autour de method() 1
autour de method() 2
après method() 1
après method() 2
```

Modifier plusieurs méthodes

Des modificateurs identiques peuvent être installés en une seule fois :

```
before qw{ method1 method2 method3 }
      => sub { say scalar localtime; };
```

Il est aussi possible de spécifier une expression régulière qui sera vérifiée avec le nom de la méthode appelée :

```
after qr/^method\d$/ => sub { say scalar
   ➥ localtime; };
```

Cependant, ces méthodes ne permettent pas de savoir quelle méthode est appelée originellement : attention donc à n'utiliser cette notation que lorsque le pré/post-traitement est strictement identique quelle que soit la méthode !

Appeler la méthode parente

super()

Dans le monde objet, il est assez facile d'écrire une classe héritant d'une autre classe. Les méthodes appelées seront alors celles de la classe fille si elles sont surchargées. `Moose` propose deux mécanismes pour que les classes de l'arbre d'héritage travaillent ensemble : appeler la méthode surchargée de la classe parente et augmenter la méthode (voir page 177).

Le premier mécanisme est courant dans les langages supportant la programmation orientée objet. Avec `Moose`, cela se fait avec un appel à la fonction `super()`. Cependant, il faut pour cela spécifier que la méthode est surchargée avec intention de faire appel à la classe de base, avec la fonction `override` :

```
package Base;
use Moose;
sub method { say 'base'; }

package Sub;
use Moose;
extends 'Base';
override method=>sub {say 'sub'; super();};

Sub->new->method;
# affiche :
#    sub
#    base
```

Comme d'habitude, la fonction `override` est importée par `Moose`. Et comme son invocation est une instruction, il faut la terminer avec un point-virgule.

La fonction `super()` ne prend pas d'arguments[2], et appelle la méthode de la classe parente avec les *mêmes* arguments que ceux de la méthode de la classe fille – même si `@_` a été modifié. La fonction `super()` renvoie la valeur de retour de la méthode parente.

Augmenter une méthode

```
inner()
augment()
```

En plus de ce fonctionnement somme toute classique dans les langages de programmation orientés objet, `Moose` propose un mécanisme astucieux de coopération entre les classes fille et parente. Ce mécanisme n'est pas courant, car c'est alors la méthode parente qui va appeller la méthode fille – à l'inverse du paradigme objet habituel (voir page 175).

Ainsi, la méthode de la classe parente va faire un appel à la fonction `inner()` importée par `Moose` :

```
package Document;
use Moose;

sub as_xml {
  my $self = shift;

  my $xml = "<document>\n";
  $xml .= inner(); # appel de la méth. fille
  $xml .= "</document>\n";

  return $xml;
}
```

2. Et ignorera donc ceux qui lui seront passés quand même…

La classe fille va surcharger la méthode de la classe parente grâce à la fonction `augment()` :

```
package Livre;
use Moose;
extends 'Document';

augment as_xml => sub {
    my $self = shift;

    my $xml = "<livre>\n";
    # ajout du contenu xml, voire
    # appel à inner()
    $xml    .= "</livre>\n";

    return $xml;
};
```

Appeler la méthode `as_xml()` sur un objet de classe `Livre` va donc appeler la méthode de la classe `Document`, qui appellera en cours de route la méthode de `Livre` :

```
say Livre->new->as_xml;
# affiche :
#   <document>
#   <livre>
#   ... contenu ...
#   </livre>
#   </document>
```

Il est bien sûr possible de continuer à appeler `inner()` dans la classe fille. C'est même recommandé, au cas où la classe fille soit elle-même sous-classée dans le futur... S'il n'y a pas de classe fille, l'appel à `inner()` ne fera rien.

Déléguer une méthode à un attribut

```
handles => . . .
```

`Moose` permet de générer des méthodes dans certains cas, afin de gagner en clarté ou en nombre de lignes de code à maintenir. Le mécanisme employé de *délégation* s'applique à un attribut ou à une structure (voir page 180).

Le premier cas consiste à créer une méthode qui va en appeler une autre sur un attribut. Cela permet de simplifier l'API de la classe, sans que les utilisateurs de la classe aient besoin de savoir comment elle est construite.

```
package Image;
use Moose;

has fichier => (
    is      => 'rw',
    isa     => 'Path::Class::File',
    handles => [ qw{ slurp stringify } ],
);
```

Ainsi, un objet `Image` correctement initialisé permettra d'écrire :

```
# contenu du fichier
my $data = $image->slurp;

# chemin du fichier image
my $path = $image->stringify;
```

Il est même possible de personnaliser l'API de la classe :

```
has fichier => (
      is      => 'rw',
```

```
      isa      => 'Path::Class::File',
      handles => {
          blob => 'slurp',
          path => 'stringify'
      },
);
```

Ce qui permet d'avoir une classe agréable à utiliser :

```
$image->blob;
# appel de $image->fichier->slurp;
$image->path;
# appel de $image->fichier->stringify;
```

Déléguer des méthodes à une structure Perl

handles => . . .

`Moose` propose de générer un certain nombre de méthodes pour les attributs d'une classe en fonction de son type. Par exemple, il est possible de créer une méthode qui fera un `push` sur un attribut de type référence de liste :

```
has liste_courses => (
      isa      => 'ArrayRef[]',
      traits   => [ 'Array' ],
      default => sub { [] },
      handles => {
          ajout    => 'push',
          suivant => 'shift',
      },
);
```

Appeler la méthode `ajout()` sur un objet de la classe ajoutera un item dans la liste `liste_courses`. Cela se fait en définissant un trait sur l'attribut.

Le tableau suivant donne la liste des traits disponibles, avec le type de l'attribut sur lequel l'appliquer et les méthodes que propose le trait.

Trait	Type	Méthodes proposées
`Array`	`ArrayRef[]`	`count`, `is_empty`, `elements`, `get`, `pop`, `push`, `shift`, `unshift`, `splice`, `first`, `grep`, `map`, `reduce`, `sort`, `sort_in_place`, `shuffle`, `uniq`, `join`, `set`, `delete`, `insert`, `clear`, `accessor`, `natatime`
`Bool`	`Bool`	`set`, `unset`, `toggle`, `not`
`Code`	`CodeRef`	`execute`, `execute_method`
`Counter`	`Num`	`set`, `inc`, `dec`, `reset`
`Hash`	`HashRef[]`	`get`, `set`, `delete`, `keys`, `exists`, `defined`, `values`, `kv`, `elements`, `clear`, `count`, `is_empty`, `accessor`
`Number`	`Num`	`set`, `add`, `sub`, `mul`, `div`, `mod`, `abs`
`String`	`Str`	`inc`, `append`, `prepend`, `replace`, `match`, `chop`, `chomp`, `clear`, `length`, `substr`

La plupart de ces méthodes sont assez faciles à comprendre et reprennent les fonctions de base à appliquer sur les hashs, tableaux, nombres et chaînes. Pour plus d'informations sur les méthodes, se reporter à la documentation des traits, `Moose::Meta::Attribute::Native::Trait::Hash` par exemple.

10

Fichiers et répertoires

La gestion de fichiers est centrale pour tout programme un tant soit peu conséquent. Les ouvrir, les lire, les écrire... Mais cela va plus loin, avec le parcours de répertoires, la portabilité entre les systèmes d'exploitation, etc. Toutes choses dont Perl se joue avec la plus grande facilité.

Ouvrir des fichiers

IO::File->new(..)

Toute manipulation de fichier en Perl passe par un descripteur de fichier. Obtenir un descripteur fichier se fait avec le constructeur de `IO::File` :

```
use IO::File;
my $path = '/chemin/vers/le/fichier';
my $fh   = IO::File->new( $path, '<' )
    or die "ne peut ouvrir '$path' : $!\n";
```

Cette méthode accepte trois arguments :

- Argument 1 : un chemin vers le fichier à ouvrir. Ce chemin doit suivre les canons du système d'application et peut être absolu ou relatif.
- Argument 2 : un mode d'ouverture. `IO::File` propose les modes d'ouverture listés dans le tableau suivant[1].

Tableau 10.1: Modes d'ouverture des fichiers

Mode	Description
<	Mode lecture
>	Mode écriture, écrase l'ancien fichier
>>	Mode écriture, concaténation après l'ancien fichier

- Argument 3 : une indication des permissions à appliquer dans le cas d'une ouverture en écriture d'un fichier qui n'existe pas.

Le constructeur renvoie un objet `IO::File` qui sert de descripteur de fichier, ou `undef` en cas d'erreur. Dans ce cas, la variable spéciale `$!` contient le code d'erreur correspondant.

Comme le constructeur de `IO::File` renvoie `undef` en cas d'erreur, il est courant de voir un appel à cette fonction suivi d'un *ou* booléen appelant la fonction `die`. Outre la concision, cela permet de produire du code qui se lit de manière naturelle : *ouvre ce fichier ou meurt*. Si les opérations à faire en cas d'erreur sont plus conséquentes, un `if` sera plus pertinent.

1. Les familiers du langage C peuvent aussi utiliser les modes d'ouverture de la fonction ANSI `fopen()` : `w`, `r`, `r+`, etc.

```
use IO::File;
my $path = '/chemin/vers/le/fichier';
my $fh   = IO::File->new( $path, '<' );
if ( not defined $fh ) {
    # traiter l'erreur
}
```

Utiliser un descripteur de fichier en lecture

getline()

Une fois le fichier ouvert, il faut bien sûr pouvoir utiliser le descripteur de fichier renvoyé par le constructeur de `IO::File`.

La lecture se fait avec la méthode `getline()`. La manière la plus simple pour lire une ligne sera donc :

```
my $line = $fh->getline;
```

La méthode `getline()` renvoyant `undef` en fin de fichier, il est donc courant de voir pour lire un fichier :

```
while ( defined( my $line = $fh->getline ) )
      {
       # utiliser $line
}
```

La méthode `getlines()` (noter le `s`) permet de lire le fichier d'un coup dans un tableau :

```
my @lines = $fh->getlines;
```

En fait, les méthodes `getline()` et `getlines()` vont utiliser la variable `$/` pour savoir ce qu'est une ligne. Une

ligne s'arrêtera après la première occurrence de cette variable. Comme `$/` est positionnée par défaut à `\n` (LF, qui vaut `\xA`) sur les systèmes de type Unix[2] et `\r\n` (`CR LF`, `\xD\xA`) pour les systèmes Windows, les méthodes `getline()` et `getlines()` renverront bien une ligne telle que le développeur s'y attend.

Il est tout à fait possible de modifier cette variable. Par exemple, pour lire un fichier paragraphe par paragraphe, il faut indiquer de lire jusqu'à rencontrer une ligne vide, c'est-à-dire deux retours chariots à la suite :

```
$/ = "\n\n";
while ( defined( my $paragraph = $fh->
    getline ) ) {
    # utiliser $paragraph
}
```

Astuce

En fait, le mode paragraphe se définit avec :

```
$/ = "";
```

Cela permettra de lire le fichier jusqu'à la prochaine ligne vide. La différence se fera en cas de plusieurs retours chariots à la suite : dans ce cas, ils seront tous considérés comme une seule ligne vide – alors qu'avec `"\n\n"` ils seront comptés comme autant de nouveaux paragraphes, même si le paragraphe ne contient qu'une ligne vide.

Pour lire le fichier d'un seul coup dans un scalaire, il suffit de positionner cette variable à `undef` :

```
undef $/;
my $data = $fh->getline;
```

2. Mac OS X est considéré comme un Unix.

Attention

La variable `$/` est utilisée dans de nombreux endroits. La modifier n'est donc pas sans impact sur le reste du code… ou des modules utilisés !

Il est donc préférable de faire ce changement dans un bloc réduit, avec le mot-clé `local` qui remet automatiquement l'ancienne valeur en place à la sortie de la portée.

```
my $data;
{
       # positionne $/ à undef de manière
       # temporaire
       local $/;
       $data = <$fh>;
}    # fin du bloc : $/ reprend
     # son ancienne valeur
```

Pour lire un fichier d'un bloc, le module `File::Slurp` facilite d'ailleurs la vie, en permettant d'écrire :

```
use File::Slurp;
my $data = read_file( '/chemin/vers/mon/
   ➥fichier' );
```

Ce qui est tout de même plus simple !

Utiliser un descripteur de fichier en écriture

```
print(..)
```

Pour écrire dans un fichier, c'est tout aussi aisé : il suffit d'utiliser la méthode `print()`.

```
$fh->print("nouvelle ligne ajoutée\n");
```

Pour être tout à fait correcte, une application se devrait de vérifier le code de retour de la fonction et d'agir en conséquence si une erreur survient :

```
$fh->print("nouvelle ligne\n")
   ➥or die "erreur lors de l'écriture : $!";
```

La méthode `printf()` est aussi disponible pour les sorties formatées. Elle prend les mêmes arguments que la fonction `printf` décrite dans `perlfunc`.

Fermer un descripteur de fichier

`close()`

Une fois les opérations de lecture ou d'écriture dans un fichier terminées, il faut fermer le fichier. Il faut utiliser la méthode `close()` et vérifier – là aussi pour une application consciencieuse – son code de retour :

```
$fh->close
   ➥or die "erreur lors de la fermeture :
   ➥$!";
```

À noter que `IO::File` va fermer le fichier automatiquement lorsque l'objet arrivera en fin de portée :

```
use IO::File;
my $path = '/chemin/vers/le/fichier';
{
  my $fh = IO::File->new( $path, '>' )
   ➥or die $!;
  print $fh "nouvelle ligne\n";
} # fermeture de $fh automatique
```

Cependant, il est préférable de fermer explicitement les fichiers : cela a le mérite de produire un code facile à comprendre, plutôt que d'avoir à deviner quand le descripteur de fichier n'est plus utilisé.

Manipuler des chemins avec Path::Class

Pour travailler avec des fichiers, il faut d'abord connaître leur emplacement dans l'arborescence du système de fichiers. De plus, travailler de manière portable n'est pas forcément facile sans utiliser le bon module. Heureusement, `Path::Class` permet de réaliser toutes ces opérations, et même plus !

```
use Path::Class;
my $file = file( 'chemin', 'sous-chemin',
   ➥'fichier.txt' );
print $file->stringify;
```

Ce court exemple donnera des résultats différents suivant la plateforme utilisée pour le test :

- `chemin/sous-chemin/fichier.txt` sous Unix ;
- `chemin\sous-chemin\fichier.txt` sous Windows.

Un objet `Path::Class` sera automatiquement converti lors d'un contexte de chaîne. Les deux notations suivantes sont donc équivalentes :

```
my $string = $dir->stringify;
my $string = "$dir";
```

Cela permet d'utiliser un objet `Path::Class` facilement dans quasiment tous les endroits qui attendent normalement une chaîne contenant un chemin vers un fichier ou un répertoire.

Pointer un fichier ou un répertoire

```
dir()
file()
```

Les constructeurs `dir()` et `file()` acceptent soit un chemin découpé logiquement (comme dans l'exemple ci-dessus), soit un chemin natif de la plateforme, soit un mélange des deux. Les exemples suivants sont tous valides et pointent sur le même fichier :

```
use Path::Class;
my $file1 = file( 'chemin', 'sous-chemin',
   ➥'file.txt' );
my $file2 = file( 'chemin/sous-chemin/
   ➥file.txt' );
my $file3 = file( 'chemin/sous-chemin',
   ➥'file.txt' );
```

Le fichier ou répertoire ainsi pointé peut ne pas exister, `Path::Class` est un module qui aide à travailler avec les chemins d'accès dans l'absolu.

Les objets vus ci-dessus pointent des chemins relatifs au répertoire courant. Pour créer des chemin absolus, il faut utiliser soit une chaîne vide en premier argument, soit la syntaxe de la plateforme :

```
use Path::Class;
my $file1 = file('', 'usr', 'bin', 'perl');
my $file2 = file('/usr/bin/perl');
```

Pointer un objet relatif

```
subdir(..)
```

Une fois un objet `Path::Class::Dir`[3] créé, il peut être utilisé pour pointer des fichiers et répertoires contenus dans ce répertoire grâce aux méthodes `file` et `subdir`.

```
use Path::Class;
my $dir = dir( '/etc' );
my $hosts = $dir->file( 'hosts' );
my $xdir  = $dir->subdir( 'X11' );
```

Les variables `$hosts` et `$xdir` seront respectivement un objet `Path::Class:File` et `Path::Class::Dir`.

Pointer un objet parent

```
parent()
```

Inversement, `Path::Class` permet de pointer le répertoire parent d'un objet avec la méthode `parent()` :

```
use Path::Class;
my $hosts = file( '/etc/hosts' );
my $xdir = dir( '/etc/X11' );
my $dir1 = $hosts->parent;
my $dir2 = $xdir->parent;
```

Les objets `$dir1` et `$dir2` pointent tous les deux sur `/etc`.

3. Un répertoire sera un objet de classe `Path::Class::Dir` tandis qu'un fichier sera un objet de classe `Path::Class::File`.

Info

À noter que cette méthode renvoie le parent logique, qui peut être différent du parent physique (en cas de lien symbolique par exemple). Si le répertoire est relatif, la notation relative du répertoire parent est utilisée.

Voici un exemple pour aider à la compréhension :

```
$dir = dir('/foo/bar');
for (1..6) {
       print "absolu : $dir\n";
       $dir = $dir->parent;
}

$dir = dir('foo/bar');
  for (1..6) {
       print "relatif : $dir\n";
       $dir = $dir->parent;
}
```

Ceci affichera sur une plateforme Unix :

```
absolu : /foo/bar
absolu : /foo
absolu : /
absolu : /
absolu : /
absolu : /
relatif : foo/bar
relatif : foo
relatif : .
relatif : ..
relatif : ../..
relatif : ../../..
```

Obtenir des informations

stat()

Une fois un objet répertoire ou fichier créé, la méthode `stat()` permet de faire un appel à la fonction du même nom. Elle renvoie un objet `File::stat`, qui possède des méthodes nommées pour accéder aux valeurs :

```
my $statf = $file->stat;
my $statd = $dir->stat;
print $statf->size;
print $statd->nlinks;
```

Créer ou supprimer un répertoire

`mkpath(..), rmtree(..)`

```
use Path::Class;
my $dir = dir( 'chemin/sous-chemin' );
$dir->mkpath;
$dir->rmtree;
```

Les méthodes `mkpath()` et `rmtree()` vont respectivement créer toute l'arborescence (cela inclut tous les répertoires intermédiaires nécessaires) et la supprimer récursivement avec tout son contenu.

Lister un répertoire

`children()`

La méthode `children()` renvoie un ensemble d'objets `Path::Class` (fichiers et répertoires) contenus dans le répertoire. Bien sûr, il est nécessaire dans ce cas que `$dir` existe et soit accessible en lecture pour pouvoir le lister :

```
use Path::Class;
my $dir = dir( 'chemin/sous-chemin' );
my @children = $dir->children;
```

Chaque membre de `@children` sera un élément de `$dir`, c'est-à-dire que les membres seront de la forme `chemin/sous-chemin/fichier.txt` et non `fichier.txt`.

La méthode `children()` ne renvoie pas les entrées correspondant aux répertoires courant et parent[4]. Pour obtenir ces entrées spéciales dans la liste des éléments d'un répertoire, il faut passer une valeur vraie au paramètre `all`.

```
# seules les entrées standard
my @children = $dir->children;

# toutes les entrées
my @children = $dir->children(all => 1);
```

Ouvrir un fichier

`open(..)`

L'opération la plus courante pour un fichier est bien sûr son ouverture. Le début de chapitre présente comment ouvrir un fichier dont l'emplacement était déjà connu. Mais la méthode `open()` d'un objet `Path::Class::File` permet de court-circuiter l'appel au constructeur de `IO::File` et de récupérer directement un descripteur sur ce fichier :

4. `.` et `..` sous Unix et Windows.

```
my $fh = $file->open($mode, $permissions);
```

Bien sûr, il faut vérifier la validité de ce descripteur qui sera `undef` en cas d'erreur, avec la variable spéciale `$!` positionnée de manière adéquate.

Comme le traitement standard d'une erreur lors de l'ouverture d'un fichier est souvent d'appeler `die` avec le message d'erreur, `Path::Class` propose deux méthodes qui font cela :

```
my $fhr = $file->openr;
my $fhw = $file->openw;
```

Elles sont équivalentes respectivement à :

```
my $fhr = $file->open('r')
   ➥or die "Can't read $file: $!";
my $fhw = $file->open('w')
   ➥or die "Can't write $file: $!";
```

Attention cependant, le message d'erreur est en anglais, et non localisé.

Finalement, la méthode `slurp()` permet de lire le fichier d'un seul coup. La méthode est sensible au contexte et renverra donc soit le contenu du fichier en mode scalaire, soit un tableau de lignes[5] en mode liste :

```
my $data  = $file->slurp;
my @lines = $file->slurp;
```

5. Lignes définies ici aussi par `$/`.

Supprimer un fichier

remove()

Supprimer un fichier peut se révéler compliqué sur certaines plateformes[6]. Pour simplifier cette opération, `Path::Class` propose la méthode `remove()` qui s'occupe des spécificités de la plateforme :

```
$file->remove;
```

Elle renvoie simplement un booléen pour indiquer le succès ou l'échec de l'opération.

Parcourir un répertoire récursivement

recurse(..)

La méthode `recurse()` permet de parcourir un répertoire de manière récursive. Pour chaque élément rencontré, la fonction anonyme passée en paramètre est appelée :

```
$dir->recurse( callback => sub { ... } );
```

Couplée à la méthode `is_dir()` qui permet de savoir si un objet rencontré est de type répertoire ou fichier, `recurse()` est un outil extrêmement puissant pour construire des fonctions de recherche performantes et concises.

L'exemple suivant permet de trouver tous les fichiers nommés `*.pm` et d'afficher leur taille :

6. Telles que VMS qui maintient des versions de fichiers.

```
$dir->recurse( callback => sub {
    my $obj = shift;
    return if $obj->is_dir;
    return if $obj =~ /\.pm/;
    print "$obj\t" . $obj->stat->size . "\n";
} );
```

Enfin, la méthode `recurse()` accepte le paramètre `depthfirst` qui permet de faire d'abord une recherche en profondeur :

```
$dir->recurse( depthfirst => 1,
                 callback => sub { ... }
               );
```

Créer un fichier temporaire

File::Temp

Créer des fichiers (ou répertoires) temporaires est une tâche courante qui est en pratique assez difficile à réaliser pour éviter les *race conditions*.

Le module `File::Temp` va créer un fichier temporaire et renverra un objet qui pourra être utilisé comme un descripteur de fichier classique. Il possède aussi une méthode `filename()` qui renvoie le nom du fichier temporaire ainsi conçu.

```
use File::Temp;
my $fh = File::Temp->new;
$fh->print( "une nouvelle ligne\n" );
print $fh->filename;
```

Le fichier temporaire va être créé dans le répertoire temporaire par défaut (souvent `/tmp` sous Unix), sauf si le paramètre `DIR` est fourni. Il est aussi possible de contrôler la forme du nom du fichier temporaire avec les paramètres `TEMPLATE` et `SUFFIX`. Le paramètre `TEMPLATE` doit comporter assez de caractères `X` qui seront remplacés aléatoirement lors de la création du fichier :

```
my $fh = File::Temp->new(
       DIR      => '/tmp/special',
       TEMPLATE => 'temp_XXXXXXXX',
       SUFFIX   => '.dat',
); # fichier créé :
   # /tmp/special/temp_YbUzIJj7.dat
```

Le fichier temporaire sera supprimé lorsque l'objet sera détruit (en fin de portée le plus souvent), mais le paramètre `UNLINK` permet de contrôler ce comportement :

```
use File::Temp;
{
     my $fh = File::Temp->new( UNLINK => 0 );

     # utilisation de $fh

} # le fichier ne sera pas supprimé
# automatiquement
```

Créer un répertoire temporaire

De la même manière, il est parfois bien pratique de créer un répertoire temporaire. Là encore, le module `File::Temp` vient à la rescousse :

```
use File::Temp;
my $tmp = File::Temp->newdir;
print $tmp->dirname;
```

Le constructeur `newdir()` comprend la même option `DIR` que pour les fichiers temporaires. Le répertoire ainsi créé sera aussi supprimé lorsque l'objet sera détruit, sauf si le paramètre `CLEANUP` est fourni avec une valeur fausse :

```
use File::Temp;
{
    my $tmp = File::Temp->newdir(
    ➥  CLEANUP => 0 );
    my $dir = $tmp->dirname;

    # utilisation de $tmp

} # le répertoire ne sera pas supprimé
  # automatiquement
```

Identifier les répertoires personnels

File::HomeDir

Certaines opérations se font sur des fichiers situés dans le répertoire personnel de l'utilisateur. Savoir où se situe ce répertoire utilisateur peut être assez compliqué suivant la plateforme, et même suivant la version de celle-ci[7]. Le module `File::HomeDir` transcende ces différences et renvoie la bonne valeur sur l'ensemble des plateformes :

```
use File::HomeDir;
my $home = File::HomeDir->my_home;
```

De plus, un certain nombre de méthodes peuvent être utilisées pour retrouver les autres répertoires personnels :

7. Par exemple, Windows XP et Windows Vista ne placent pas le répertoire personnel des utilisateurs au même endroit.

```
use File::HomeDir;
my $desktop = File::HomeDir->my_desktop;
   ➥# bureau
my $docs    = File::HomeDir->my_documents;
   ➥# documents
my $music   = File::HomeDir->my_music;
   ➥# musique
my $pics    = File::HomeDir->my_pictures;
   ➥# photos
my $videos  = File::HomeDir->my_videos;
   ➥# films
my $data    = File::HomeDir->my_data;
   ➥# données
```

Dans le monde Unix, toutes sortes de données sont mélangées dans le répertoire personnel – bien que cela soit en train de changer sur les plateformes supportant la norme FreeDesktop. Mais les autres plateformes utilisent depuis quelques temps des répertoires différents pour stocker des données différentes. Une application utilisant `File::HomeDir` devra donc essayer de recourir à la méthode la plus spécifique possible : les documents utilisateurs devront être sauvegardés dans `my_documents()`, les données internes d'une application non destinées à l'utilisateur dans `my_data()`. Ceci ne prêtera pas à conséquence sur les plateformes ne faisant pas de distinction, car `File::HomeDir` renverra le répertoire personnel par défaut pour ces répertoires particuliers. Mais cela permettra à l'application de fonctionner de manière native sur les plateformes pour qui cette distinction est importante.

La valeur renvoyée par ces diverses méthodes est une chaîne, à compléter d'un appel à `Path::Class` pour profiter des facilités de ce module.

```
use File::HomeDir;
use Path::Class;
my $home = dir( File::HomeDir->my_home );
```

Enfin, il est bon de noter que ces fonctions s'assureront que les répertoires renvoyés existent. Ainsi, l'application peut être sûre qu'aucune *race condition* n'arrivera lors de leur utilisation.

Connaître le répertoire courant

getcwd()

Beaucoup de programmes travaillent avec des fichiers qui par défaut seront relatifs au répertoire courant. Il est donc important de savoir comment utiliser ce répertoire en Perl.

```
use Cwd;
my $dir = getcwd;
```

Récupérer le répertoire courant se fait avec la fonction `getcwd()` du module `Cwd`. La valeur renvoyée est une chaîne, qu'il faut donc traiter pour pouvoir utiliser la puissance de `Path::Class` (voir page 189) :

```
use Cwd;
use Path::Class;
my $dir = dir( getcwd );
```

Changer de répertoire

chdir()

La fonction `chdir()` connue des programmeurs C existe aussi en Perl, et est sans doute le moyen le plus facile pour changer de répertoire :

```
chdir $newdir;
```

Cependant, l'inconvénient de cette fonction est que l'ancien répertoire est, par définition, complètement oublié. Le module `File::pushd` permet de pallier ce problème, en empilant les répertoires de la même manière que son équivalent `pushd` du shell *bash*[8].

```
use File::HomeDir;
use File::pushd;

chdir File::HomeDir->my_home;
{
       # changement de répertoire durant
       # une portée limitée
       my $dir = pushd( '/tmp' );

       # travail dans /tmp

}  # fin de la portée :
   # retour au répertoire home
```

Comme il est courant de changer de répertoire pour un répertoire temporaire, `File::pushd` propose aussi la fonction `tempd()` qui combine `pushd()` avec un appel à `File::Temp`.

```
use File::pushd;
{
       my $dir = tempd();

       # travail dans un répertoire
           temporaire

}  # retour au répertoire initial
```

Le répertoire temporaire sera automatiquement supprimé en fin de portée.

8. La fonction `popd` n'existe pas, la fin de portée joue ce rôle.

11

Bases de données SQL

Est-il besoin de préciser à quel point les bases de données sont omniprésentes dans le monde informatique, sous des formes très diverses et avec des paradigmes très différents en fonction des besoins ? Ce chapitre est consacré aux bases de données relationnelles car elles représentent un passage quasi obligé dans la programmation actuelle ; mais il faut garder à l'esprit qu'elles ne représentent qu'un aspect de l'univers, plus large qu'il n'y paraît, des nombreux autres paradigmes de bases de données existant depuis longtemps (les fichiers */etc/passwd* d'Unix, la base de registres de Windows, le DNS) ou apparus plus récemment (CouchDB, TokyoCabinet, Redis, etc.).

Bien évidemment, Perl dispose de tout ce qui est nécessaire pour se connecter aux bases de données. De manière amusante, alors que le slogan de Perl est « il y a plus d'une manière de le faire », dans le cas des bases de données, le module DBI règne sur les mécanismes de connexion, établissant le standard de fait.

DBI (*Database Interface*) constitue véritablement une API générique et normalisée d'accès aux bases de données (dans un sens très large du terme), comparable dans l'esprit à

JDBC dans le monde Java. Les mécanismes de connexion et le protocole proprement dits sont implémentés dans les pilotes correspondants à chaque base, les DBD *Database Driver*.

`DBI` est un de ces modules majeurs dans le monde Perl qui ont conduit les développeurs à contribuer à de nombreux pilotes sur CPAN, et comme souvent quand il y a pléthore, certains sont corrects, certains bien moins, d'autres encore sont très bons. Il ne sera pas surprenant de trouver dans cette dernière catégorie les pilotes des bases majeures que sont Oracle, PostgreSQL, MySQL, DB2, Ingres, SQLite. Un pilote ODBC générique est disponible pour les bases Windows.

Info

Une dernière catégorie de pilotes mérite l'attention : ceux écrits par des gens pouvant être qualifié de « dérangés », mais qui peuvent s'avérer être la seule solution dans bien des situations compliquées. Il s'agit des pilotes écrits en pur Perl, permettant de se connecter à certaines bases sans utiliser leurs bibliothèques usuelles. Sans surprise, ces pilotes n'existent que pour les bases libres PostgreSQL et MySQL : `DBD::PgPP` pour PostgreSQL, et `DBD::mysqlPP` et `DBD::Wire10` pour MySQL (`DBD::Wire10` pouvant aussi se connecter aux bases Drizzle et Sphinx). Certes, ces pilotes seront moins efficaces que les versions natives, mais ils peuvent être salvateurs s'il faut déployer dans un environnement ancien ou très contraignant (conflits de versions des clients). À noter qu'il existe de même une version pur Perl de DBI, par défaut nommée `DBI::PurePerl`.

Enfin, il faut savoir que `DBI` fonctionne de manière transparente par rapport au langage de requêtage. S'il s'agira de SQL dans la très grande majorité des cas, il est parfaitement possible d'écrire des pilotes avec un autre langage. Ainsi, `DBD::WMI` utilise logiquement WQL (*WMI Query*

Language) pour interroger la base WMI sous Windows, et l'un des auteurs travaille sur `DBD::Nagios`, qui utilise LQL (*Livestatus Query Language*).

S'il est encore besoin de montrer la capacité d'adaptation de `DBI` aux environnements les plus exigeants, il suffit de mentionner qu'il est fourni avec deux modules de proxy différents, l'un, `DBD::Proxy`, avec conservation d'état (*stateful*), le second, `DBD::Gofer`, sans conservation d'état (*stateless*). Cas d'utilisation typiques : traversée de pare-feu ou connexion à une ou plusieurs bases exotiques dont les pilotes ne sont disponibles qu'en JDBC. `DBD::Gofer` peut aussi servir de système de répartition de charge.

Se connecter

`DBI->connect()`

```
use DBI;

my $dbh = DBI->connect(
      "dbi:Pg:host=dbhost;dbname=mybase",
      $login, $password, { RaiseError => 1 }
);

$dbh->do("TRUNCATE logs");
$dbh->prepare("SELECT id, name
   ➥FROM customers");
# ...
```

La méthode `connect()` de `DBI` est un constructeur, similaire à la méthode `new()` d'autres modules, qui tente d'établir la connexion à la base et renvoie un objet descripteur de base (DBH, *database handler*).

Les trois premiers arguments sont la chaîne de connexion (ou DSN, *data source name*), le nom d'utilisateur et le mot de passe. Ces deux derniers peuvent être des vides, mais il faut quand même les fournir. La chaîne de connexion ressemble à une URL et se décompose ainsi :

- `"dbi:"` : La chaîne `"dbi:"` est l'équivalent du protocole et définit ici l'API ; elle est donc toujours celle de DBI.
- **Le nom du module**. Le nom du module Perl qui fournit le pilote à utiliser, sans son préfixe `DBD::`, ainsi pour accéder à une base Oracle, le module à utiliser est `DBD::Oracle`, d'où le nom `Oracle`. Idem pour SQLite. Par contre, pour PostgreSQL le module Perl s'appelle `DBD::Pg`, et pour MySQL, `DBD::mysql` (tout en minuscules). Si aucun nom n'est fourni, DBI ira le chercher dans la variable d'environnement `$DBI_DRIVER`. À noter que DBI chargera automatiquement le module correspond.
- **Les arguments**. Le reste de la chaîne, après les deux-points qui suit le pilote, constitue les arguments du pilote, et n'est pas interprété par DBI. Si la chaîne fournie est vide, `connect()` ira la chercher dans la variable d'environnement `$DBI_DSN`.

Le dernier argument de `connect()`, optionnel, est une référence vers un hash contenant des attributs DBI. Parmi les nombreux attributs existants, il n'est vraiment besoin que de connaître les suivants :

- `AutoCommit` : active ou désactive le support des transactions. La valeur vraie, positionnée par défaut en respect des conventions JDBC et ODBC, désactive le support des transactions. Il faut donc explicitement positionner cet attribut à faux pour activer les transactions, si le pilote les supporte. Les bonnes pratiques conseillent de toujours le positionner à la valeur attendue par le programme, même s'il s'agit de la valeur par défaut.

- `RaiseError` : force les erreurs à provoquer des exceptions, plutôt que de simplement renvoyer des codes d'erreur. Par défaut désactivée, cette option est très utile pour l'écriture de courts programmes où la gestion des erreurs rajoute une quantité non négligeable de code assez ennuyeux.
- `ShowErrorStatement` : permet d'afficher la requête qui a provoqué l'erreur dans les messages.
- `TraceLevel` : c'est un des moyens pour activer les traces d'exécution.
- `Profile` : c'est un des moyens pour activer le profilage d'exécution.

Du point de vue de l'utilisateur, un des aspects gênants de la chaîne de connexion, et en particulier des arguments du pilote, est que ceux-ci sont spécifiques à chaque pilote, et qu'il n'y a donc aucune normalisation alors que, dans bien des cas, les paramètres sont les mêmes : nom de la base, éventuels nom d'hôte et numéro de port. Pour ces raisons, l'un des auteurs a écrit un petit module, `DBIx::Connect::FromConfig`, qui offre une interface un peu plus homogène :

```
use DBI;
use DBIx::Connect::FromConfig -in_dbi;

my %settings = (
      driver       => "Pg",
      host         => "bigapp-db.society.com",
      database     => "bigapp",
      username     => "appuser",
      password     => "sekr3t",
      attributes   => { AutoCommit => 1,
        ➥     RaiseError => 1 },
);
```

```
my $dbh = DBI->connect_from_config(
   ➥config => \%settings );
```

C'est déjà intéressant, mais mieux encore, le module peut prendre l'objet d'un module de configuration en argument :

```
my $config = Config::IniFiles->new(
   ➥-file => "/etc/myprogram/database.conf"
);
my $dbh = DBI->connect_from_config(
   ➥config => $config);
```

où le fichier de configuration ressemble à :

```
[database]
driver      = Pg
host        = bigapp-db.society.com
database    = bigapp
username    = appuser
password    = sekr3t
attributes  = AutoCommit=1¦RaiseError=1
```

Ce qui simplifie la gestion de la configuration lorsqu'une application doit être déployée dans des environnements différents (développement, recette, préproduction, production).

Tester la connexion

```
$dbh->ping()
```

Bien que ce soit rarement nécessaire, il est possible de tester si la connexion est toujours active (si ce mécanisme est véritablement codé dans le pilote correspondant).

Se déconnecter

```
$dbh->disconnect()
```

Se déconnecter d'une base ne demande qu'un appel à la méthode `disconnect()`. À noter qu'elle est automatiquement invoquée quand le descripteur de base est sur le point d'être détruit, par exemple à la fin du bloc où il a été défini ; mais il est plus propre de l'appeler explicitement. Un avertissement sera affiché si des requêtes étaient encore actives, par exemple une requête `SELECT` avec des données non encore récupérées.

Préparer une requête SQL

```
$dbh->prepare(. . .)
```

La manière la plus efficace d'utilisation des requêtes est de suivre le protocole en quatre étapes : *prepare* (préparation), *bind* (liaison), *execute* (exécution), *fetch* (récupération).

La *préparation* consiste en l'analyse de la requête, soit au niveau du client (ici, du pilote), soit au niveau du serveur, afin de vérifier que la syntaxe est correcte, et quand il y en a de repérer les paramètres. En effet, `DBI` supporte le passage de valeurs dans les requêtes au travers de marqueurs (*placeholders*), notés par des points d'interrogation :

```
INSERT INTO users (login, name, group)
  ➥VALUES (?, ?, ?)
```

Comme le montre l'exemple, les marqueurs indiquent l'emplacement de chaque valeur, et ne doivent donc pas

être mis entre guillemets, même si la valeur qui sera passée est une chaîne. Un marqueur ne correspond qu'à une seule et unique valeur, ce qui interdit de construire une requête avec un `IN (?)` en espérant passer plus d'une valeur ; il est nécessaire d'expliciter chaque valeur par un marqueur. La solution pour ce genre de cas est de construire la requête dynamiquement :

```
my $list  = join ", ", ("?") x @values;
my $query = "SELECT * FROM library WHERE
   ➥author IN ($list)";
```

Attention

Les marqueurs ne peuvent correspondre qu'à des valeurs, et non à des noms de table ou de colonne (voir page 223 pour une solution).

La méthode `DBI` pour préparer une requête s'appelle tout simplement `prepare()` ; elle s'invoque sur un objet DBH et renvoie un descripteur de requête (STH, *statement handler*) :

```
my $sth = $dbh->prepare(
 "SELECT * FROM library WHERE author LIKE ?"
);
```

Valeurs nulles en SQL

Un point important à noter est que les valeurs `NULL` (au sens SQL) sont traduites en Perl par `undef`. Cela marche aussi bien en entrée qu'en sortie, mais il faut se souvenir qu'en SQL, `value = NULL` doit s'écrire `value IS NULL`, ce qui nécessite de traiter différemment les arguments non définis.

Lier une requête SQL

```
$str->bind_param(. . .)
```

Après avoir préparé une requête, les valeurs sont associées aux marques lors de l'étape de *liaison*, à l'aide de la méthode `bind_param()` :

```
$sth->bind_param(1, "Naoki Urasawa");
```

Le premier argument est le numéro du marqueur (ceux-ci sont numérotés à partir de 1), le deuxième la valeur. Il est aussi possible de passer en troisième argument le type SQL de la valeur :

```
$sth->bind_param(1, "Naoki Urasawa",
  ➥SQL_VARCHAR);
```

En pratique, cette manière de faire est rarement utilisée car la liaison peut être réalisée en même temps que l'*exécution*.

Exécuter une requête SQL

```
$str->execute(. . .)
```

Il est possible d'exécuter une requête en liant les paramètres en un seul appel à la fonction `execute` :

```
$str->execute("Naoki Urasawa");
```

Récupérer les données de retour d'une requête SQL

La *récupération* des enregistrements se fait à l'aide des méthodes `fetch`*`xxx_yyy`*`()`. La partie *xxx* du nom correspond à la manière de récupérer les enregistrements, soit un par un (`row`), soit tout d'un coup (`all`). La partie *yyy* du nom est la forme sous laquelle `DBI` va fournir les données : tableau direct (`array`), référence de tableau (`arrayref`), référence de hash (`hashref`).

Ainsi, avec la requête précédente, un appel à `fetchrow_array()` donnera :

```
my @row = $sth->fetchrow_array();
# @row = ( "PLUTO volume 1", "2010-02-19",
#          "Naoki Urasawa", "Kana" )
```

`@row` reçoit les valeurs du premier enregistrement, dans l'ordre des colonnes indiqué dans le `SELECT` ou dans l'ordre naturel (typiquement, celui de création). Même chose avec `fetchrow_arrayref()` :

```
my $row = $sth->fetchrow_arrayref();
# $row = [ "PLUTO volume 1", "2010-02-19",
#          "Naoki Urasawa", ... ]
```

`fetchrow_hashref()` renvoie l'enregistrement sous forme d'une référence de hash, avec les colonnes en clés :

```
my $row = $sth->fetchrow_hashref();
# $row = { title => "PLUTO volume 1",
#          date => "2010-02-19",
#          author => "Naoki Urasawa", ... }
```

Bien sûr, les appels successifs à ces méthodes permettent de récupérer les enregistrements au fur et à mesure ; elles renvoient faux quand il n'y a plus d'enregistrement :

```
while (my $book = $sth->fetchrow_hashref()){
 print "* $book->{title} ($book->{date})\n";
}
# affiche :
# * PLUTO volume 1 (2010-02-19)
# * PLUTO volume 2 (2010-02-19)
# * PLUTO volume 3 (2010-04-02)
# ...
```

`fetchall_arrayref()` renvoie assez simplement tous les enregistrements dans une référence de tableau englobant :

```
my $books = $sth->fetchall_arrayref();
# $books = [
# [ "PLUTO volume 1", "2010-02-19",
   ➥"Naoki Urasawa" ],
# [ "PLUTO volume 2", "2010-02-19",
   ➥"Naoki Urasawa" ],
# [ "PLUTO volume 3", "2010-04-02",
   ➥"Naoki Urasawa" ],
# ...
# ]
```

`fetchall_hashref()` est un peu plus subtile. Elle attend comme argument le nom d'une colonne qui fait office de clé, c'est-à-dire d'identifiant unique de chaque enregistrement. Cela permet ainsi d'accéder à chaque champ de manière directe :

```
my $library=$sth->fetchall_hashref("title");
# $library = {
#   "PLUTO volume 1" => {
#     date => "2010-02-19", author =>
  ➥ "Naoki Urasawa",
#   },
#   "PLUTO volume 2" => {
#     date => "2010-02-19", author =>
```

```
   ➥  "Naoki Urasawa",
#    },
#    "PLUTO volume 3" => {
#      date => "2010-04-02", author =>
   ➥  "Naoki Urasawa",
#    },
#    ...
# }
```

S'il n'y a pas une colonne pour identifier de manière unique chaque enregistrement, une référence vers un tableau de clés peut être fournie à la méthode, qui construira le hash résultat avec autant de niveaux de profondeur que de clés.

```
my $sth = $dbh->prepare(q{
    SELECT  host, service, id, type,
              level, time, message
    FROM    events
    WHERE   type = ?  AND  level = ?
});

$sth->execute("persistent", "error");
my $events = $sth->fetchall_hashref(
                 ["host", "service", "id"]
               );

say "from $host: . $events->{$host}
   ➥{$service}{$id}{message}";
```

Combiner les étapes d'exécution d'une requête SQL

Dans le cas particulier des requêtes `SELECT`, il existe des méthodes permettant de combiner les quatre étapes en une seule : `selectrow_array()`, `selectrow_arrayref()`, `selectrow_hashref()`, `selectall_arrayref()` et `selectall_hashref()`.

```
my @row = $dbh->selectrow_array(
      "SELECT name FROM users WHERE id = ?",
      ➥     {}, $id
);
```

Le second argument de la méthode est une référence vers un hash, vide dans l'exemple, qui accepte des attributs (comme `RaiseError`) qui seront positionnés juste pour cette requête.

Pour les requêtes, hors `SELECT`, qui n'ont pas besoin d'être exécutées de nombreuses fois, la méthode `do()` permet de tout faire d'un coup :

```
$dbh->do("TRUNCATE logs");
```

Bien sûr, les requêtes peuvent contenir des marqueurs, les valeurs étant passées de manière similaire aux méthodes `selectxxx()` :

```
my $n = $dbh->do(
      "DELETE FROM events WHERE host=? AND
      ➥     service=?",
      {}, $host, $service,
);
```

La méthode renvoie le nombre de lignes affectées par la requête, `undef` en cas d'erreur et -1 si le nombre n'est pas connu ou n'a pas de sens.

Cette mécanique peut sembler un peu lourde, mais elle permet en réalité d'obtenir de très bonnes performances, en particulier quand les méthodes les plus rapides (comme `fetchrow_arrayref()`) sont utilisées. `DBI` dispose d'autres mécanismes pour gagner encore en performance, comme par exemple `bind_col()`, mais ils ne seront pas abordés car les quelques méthodes présentées ici permettent déjà d'obtenir de très bons résultats.

Mise en cache et sécurité

Un point fondamental à comprendre est que la préparation d'une requête autorise bien plus facilement la mise en cache de celle-ci, ce qui permet de la réutiliser par la suite plus rapidement. L'utilisation des marqueurs est de ce point de vue essentielle pour deux raisons. D'une part, cela garantit que le texte de la requête ne change pas, et donc qu'il s'agit bien de la même requête. D'autre part, l'interpolation des valeurs conduit immanquablement à des problèmes de sécurité, les fameuses injections SQL. Il n'existe aucun moyen fiable d'exécuter des requêtes avec des données interpolées. La seule manière est justement de remplacer les valeurs par des marqueurs, et d'établir *a posteriori* une correspondance entre chaque marqueur et sa valeur.

Étant donné que ce principe offre à la fois sécurité et rapidité, les quelques rares inconvénients qui peuvent exister sont plus que largement compensés par la tranquillité d'esprit qu'il procure.

Gérer les erreurs

```
$h->err $h->errstr $DBI::err $DBI::errstr
```

Pour vérifier si une erreur s'est produite (si `RaiseError` n'est pas utilisé), DBI fournit la méthode `err()` qui du point de vue utilisateur final renvoie un booléen (en pratique, il s'agit d'un code d'erreur, mais dont la signification n'a généralement de sens que pour le pilote). Pour avoir le détail de l'erreur, c'est la méthode `errstr()` qui doit être utilisée. Les valeurs de ces méthodes ont une du-

rée de vie assez courte, car elles sont remises à zéro par un appel à la plupart des autres méthodes.

```
if ($sth->err) {
       die "erreur: ", $sth->errstr;
}
```

Il existe aussi des variables globales `$DBI::err` et `$DBI:errstr` qui ont pour valeur celles des méthodes `err()` et `errstr()` du dernier descripteur qui a été utilisé. Il est sans surprise déconseillé de les utiliser et il faut leur préférer les méthodes correspondantes.

Tracer l'exécution

```
DBI_TRACE $h->{TraceLevel} trace()
```

`DBI` intègre un mécanisme pour tracer l'exécution des requêtes afin de faciliter la recherche des problèmes. Celui-ci peut s'activer de plusieurs manières :

- par la variable d'environnement `$DBI_TRACE` ;
- par l'attribut `TraceLevel` ;
- par la méthode `trace()`.

La valeur à passer est le niveau de verbosité souhaité de la trace. En pratique, la valeur 2 est la plus appropriée pour commencer car elle affiche l'ensemble des valeurs passées en argument. Ainsi, le code suivant :

```
$dbh->trace(2);

my $sth = $dbh->prepare(q{
       INSERT INTO events (host, service, id,
       ➥      message)
       VALUES (?, ?, ?, ?)
});
```

```
$sth->execute('ns.domain.com', 'dns',
   ➥      123, 'dns system ok'
);
```

affiche cette trace :

```
DBI::db=HASH(0x186ba18) trace level set to 0x0/2 (DBI @ 0x0/0)
in DBI 1.609-ithread (pid 7346)
-> prepare for DBD::SQLite::db (DBI::db=HASH(0x186ba60)~0x186ba18 '
INSERT INTO events (host, service, id, message) VALUES (?, ?, ?, ?)
') thr#1800400
<- prepare= DBI::st=HASH(0x1847a70) at dbi-trace line 15
-> execute for DBD::SQLite::st (DBI::st=HASH(0x1847a70)~0x186bc88
'ns.domain.com' 'dns' 123 'dns system ok') thr#1800400
<- execute= 1 at dbi-trace line 18
-> DESTROY for DBD::SQLite::st (DBI::st=HASH(0x186bc88)~INNER)
thr#1800400
<- DESTROY= undef
-> DESTROY for DBD::SQLite::db (DBI::db=HASH(0x186ba18)~INNER)
thr#1800400
<- DESTROY= undef
```

La variable d'environnement présente l'avantage d'activer les traces sans modifier le programme :

```
DBI_TRACE=2  perl dbi-program ...
```

Par contre, l'attribut permet d'activer la trace de manière locale à un bloc :

```
{
    # seules les requêtes créées dans ce
        bloc
    # seront tracées
    local $dbh->{TraceLevel} = 2;
    ...
}
```

Cet attribut est comme beaucoup d'autres hérité par les requêtes du descripteur de base à partir duquel elles sont créées :

```
my $sth;

{
      local $dbh->{TraceLevel} = 2;
      $sth = $dbh->prepare("...");
}

# cet appel sera tracé en sortie
$sth->execute;
```

La méthode permet de rediriger la trace, par défaut affichée sur la sortie d'erreur, vers un fichier ou un descripteur de fichier :

```
$fh = IO::File->new(">trace.log");
$h->trace(2, $fh);
```

Profiler l'exécution

```
DBI_PROFILE $h->{Profile}
```

`DBI` intègre un profileur très abouti, disposant de plusieurs interfaces pour y accéder. Seule la première sera véritablement présentée ici mais elle convient à la plupart des besoins. Un peu comme avec les traces, le profileur peut se contrôler d'une part avec la variable d'environnement `$DBI_PROFILE`, d'autre part avec l'attribut `Profile`. Dans les deux cas, il suffira d'affecter une valeur décrivant le profilage à effectuer. Dans sa forme la plus simple, il s'agira d'une valeur numérique, en particulier 2 qui affiche un rapport indiquant le coût et l'utilisation de chaque requête exécutée :

```
$ DBI_PROFILE=2  perl init-database.pl
DBI::Profile: 0.230871s 23.09% (47 calls) dbi-profile @ 2010-08-30
```

```
 02:42:09
  '' => 0.007850s / 13 = 0.000604s avg (first 0.007669s, min
  0.000004s, max 0.007669s)
'CREATE TABLE events ( host varchar, service varchar, id integer,
  message varchar )' =>
  0.052813s
'INSERT INTO events (host, service, id, message) VALUES
  (?, ?, ?, ?)' => 0.170209s / 33 = 0.005158s avg (first 0.000345s,
   min 0.000043s, max 0.017538s)
```

Les valeurs indiquées pour la requête vide correspondent en réalité au temps passé dans les méthodes propres au descripteur de base de données (connexion, affectation des attributs, etc). Pour les autres requêtes, cela correspond au temps total passé que chacune a consommé. Dans le cas du `INSERT`, la requête a été préparée une fois puis exécutée 30 fois, ce qui signifie que le profileur a aussi compté deux appels internes. Pour avoir plus de détails, il suffit d'augmenter la valeur passée à `$DBI_PROFILE` à 6, qui affiche pour chaque requête le temps passé dans chaque méthode :

```
'INSERT INTO events (host, service, id, message) VALUES
  (?, ?, ?, ?)' => 'DESTROY' =>
    0.002115s / 2 = 0.001058s avg (first 0.000043s, min 0.000043s,
     max 0.002072s)
  'execute' =>
    0.331492s / 30 = 0.011050s avg (first 0.011977s, min 0.004121s,
     max 0.057116s)
  'prepare' =>
     0.000341s
```

Le profileur de base accepte une configuration plus poussée au travers d'arguments comme `!File`, qui groupe les résultats par nom de fichiers, ou `!Time~`*N*, qui permet de grouper les résultats par tranche de *N* secondes :

```
$ DBI_PROFILE='!File:!Time~2:!Statement' perl init-database.pl
DBI::Profile: 1.437674s 143.77% (317 calls) dbi-profile
@ 2010-08-30 03:26:14
  '1283131572' => '' =>
```

```
  0.007492s / 12 = 0.000624s avg (first 0.007335s, min 0.000003s,
     max 0.007335s)
'INSERT INTO events (host, service, id, message) VALUES
  (?, ?, ?, ?)' =>
    0.490690s / 85 = 0.005773s avg (first 0.000342s, min 0.000342s,
       max 0.033933s)
'CREATE TABLE events ( host varchar, service varchar, id integer,
  message varchar )' =>
     0.053218s
  '1283131574' => '' =>
     0.000024s
'INSERT INTO events (host, service, id, message) VALUES
  (?, ?, ?, ?)' =>
   0.886250s / 218 = 0.004065s avg (first 0.003718s, min 0.000041s,
     max 0.029571s)
```

Seule la surface du profileur PERMOD:DBI a été abordée. Même l'interface « simple » offre déjà une configuration assez poussée au travers de l'objet `DBI::Profile` qui peut être affecté à l'attribut `Profile`. L'utilisateur avec des besoins plus avancés se tournera vers `DBI::ProfileDumper`, qui permet d'enregistrer les résultats du profilage sur disque, et la commande `dbiprof` pour les exploiter.

En reprenant l'exemple précédent, les commandes suivantes permettent de voir quelle est la requête la plus consommatrice :

```
$ DBI_PROFILE=6/DBI::ProfileDumper perl init-database.pl
$ dbiprof --sort count --number 1

DBI Profile Data (DBI::ProfileDumper 2.009894)

  Program       : pad/dbi-profile
  Path          : [ !Statement, !MethodName ]
  Total Records : 9 (showing 1, sorted by count)
  Total Count   : 317
  Total Runtime : 2.252447 seconds

#####[ 1 ]#########################################
  Count         : 300
  Total Time    : 1.976474 seconds
```

```
Longest Time  : 0.204763 seconds
Shortest Time : 0.002353 seconds
Average Time  : 0.006588 seconds
Key 1         : INSERT INTO events (host, service,
                id, message) VALUES (?, ?, ?, ?)
Key 2         : execute
```

12

Abstraction du SQL, ORM et bases non-SQL

`DBI` (voir page 203) permet l'exécution de requêtes paramétrées, autorisant un passage de valeurs de manière sûre, sans risque d'injection SQL. Mais comme il faut passer les valeurs dans l'ordre de définition des paramètres, les requêtes deviennent rapidement peu confortables à gérer au-delà de trois ou quatre paramètres. Certains pilotes autorisent l'utilisation de marqueurs nommés, mais ils sont rares, et forcément ce n'est pas portable.

Pour contourner cette limitation, il faut passer par un module d'abstraction du SQL. Parmi les différents disponibles sur CPAN, il en existe un qui constitue un bon compromis entre abstraction et simplicité d'utilisation, `Data::Phrasebook::SQL`.

Utiliser `Data::Phrasebook::SQL`

Il s'agit d'un module faisant partie de `Data::Phrasebook`, qui est un système de gestion de dictionnaires. Le principe est de rassembler en un seul endroit des chaînes de texte

qui peuvent apparaître à plusieurs emplacements du code, par exemple les messages d'erreurs. Si cela paraît confus, pensez aux bibliothèques comme `gettext` : au final, il s'agit de récupérer la chaîne correspondant à un identifiant (généralement la chaîne en anglais) et à la langue destination, toutes les chaînes étant stockées dans les fameux fichiers *.po*, qui constituent autant de dictionnaires. Par rapport à `gettext`, l'intérêt de `Data::Phrasebook` est qu'il supporte de nombreux formats de fichiers, et surtout autorise des variables de remplacement, un peu comme dans un système de *templates*.

`Data::Phrasebook::SQL` est donc une sous-classe de `Data::Phrasebook` spécialisée dans la gestion de dictionnaires SQL et dans l'exécution de ces requêtes au travers de `DBI`. Le gros intérêt est que cela permet une séparation entre le code Perl et le code SQL, ce qui est généralement considéré comme une bonne pratique.

Une combinaison intéressante est d'utiliser YAML comme format de stockage, car il permet d'écrire les requêtes de manière aérée et naturelle. Le cas d'utilisation qui va servir d'exemple est adapté d'une situation rencontrée par l'un des auteurs, qui sans l'aide de `Data::Phrasebook::SQL` aurait dû écrire des dizaines et des dizaines de requêtes SQL toutes semblables. Le seul vrai défaut de ce module est de pécher au niveau de la documentation, ce qui rend son apprentissage moins aisé qu'il ne devrait être.

À l'utilisation, le module s'avère par contre véritablement agréable, car il s'agit au final d'une surcouche assez fine au-dessus de DBI, qui ne perturbe pas l'utilisateur habitué à ce dernier. `Data::Phrasebook::SQL` attend d'ailleurs de l'utilisateur qu'il lui fournisse le descripteur de base, l'instanciation à partir des requêtes stockées dans un fichier YAML ressemblant à ceci :

```
my $book = Data::Phrasebook->new(
  class  => "SQL",   dbh  => $dbh,
  loader => "YAML",  file => "queries.yaml",
);
```

Pour planter le contexte, considérons une base de données assez classique avec une table qui contient des informations sur des serveurs :

```
CREATE TABLE hosts (
      host_id     integer,    -- ID du serveur
      host_name   varchar,    -- nom du serveur
      hard_id     integer,    -- ID du matériel
      site_id     integer,    -- ID du site
      cust_id     integer,    -- ID du client

      PRIMARY KEY (host_id)
);
```

En DBI classique, la requête pour trouver l'ID d'un serveur à partir de son nom s'écrit :

```
my $sth = $dbh->prepare(
    "SELECT host_id FROM hosts WHERE
    ➥  host_name = ?"
);
$sth->execute($name);
my ($id) = $sth->fetchrow_array;
```

Avec Data::Phrasebook::SQL, la requête est stockée dans un fichier, ici YAML, et est donc nommée :

```
get_host_id: ¦
      SELECT  host_id
      FROM    hosts
      WHERE   host_name = :host_name
```

La barre verticale qui suit le deux-points est la syntaxe YAML qui signale que le bloc de lignes qui suit, jusqu'à la prochaine ligne vide, est à affecter comme valeur à

`get_host_id`. `:host_name` est la syntaxe `Data::Phrasebook` pour écrire un marqueur nommé `host_name`.

Côté Perl, le code devient :

```
my $qry = $book->query("get_host_id");
$qry->execute(host_name => $host_name);
my ($host_id) = $qry->fetchrow_array;
```

Par rapport à DBI, la requête est récupérée et préparée en indiquant simplement son nom, qui fournit un objet comparable au descripteur de requête de DBI. Par contre, sa méthode `execute()` reçoit maintenant les valeurs sous forme paramétrée, attendant comme noms ceux des marqueurs de la requête. La récupération des résultats est quant à elle identique au cas DBI.

Pour les besoins de l'application et de l'exemple, considérons qu'il est nécessaire de pouvoir accéder à chaque champ de manière indépendante. Avec DBI, il faut alors écrire une requête pour chaque champ. Avec `Data::Phrasebook::SQL`, ce n'est qu'une question de remplacement de marqueurs. Ainsi, côté SQL :

```
get_host_field: ¦
      SELECT  :host_field
      FROM    sys_hosts
      WHERE   host_id = :host_id
```

et côté Perl :

```
my $qry = $hosts->query(
    "get_host_field",
    replace => { host_field => "hard_id" },
);

$qry->execute(host_id => $host_id);
my ($hard_id) = $qry->fetchrow_array;
```

Le seul ajout est celui du `replace` dans l'appel de `query()` qui indique les marqueurs à substituer pour générer la requête SQL (ce qui constitue la partie `templating`), les marqueurs restants étant remplacés lors de l'appel à `execute()` (ce qui constitue la partie paramètres nommés).

Si ce mécanisme semble trop lourd pour répondre à un besoin ici assez simple, il suffit de corser un peu la donne en ajoutant quelque chose de typique dans les grosses bases de données : un système de propriétés génériques.

```
CREATE TABLE prop_list (
  prop_id     integer, -- ID de la propriété
  prop_name   varchar, -- nom de la propriété
  prop_type   varchar, -- type de la propriété

  PRIMARY KEY (prop_id)
);

CREATE TABLE prop_values (
  val_id      integer, -- ID de la valeur de propriété
  val_prop_id integer, -- ID de la propriété
  val_host_id integer, -- ID de l'hôte associé

  val_t_bool  boolean, -- champ pour valeur booléenne
  val_t_int   integer, -- champ pour valeur entière
  val_t_float float,   -- champ pour valeur flottante
  val_t_char  varchar, -- champ pour une courte chaîne
  val_t_text  text,    -- champ pour du texte long
  val_t_date  date,    -- champ pour une date

  PRIMARY KEY (val_id)
);
```

Avec `Data::Phrasebook::SQL`, écrire les requêtes pour gérer cela est confondant de simplicité. Il suffit en effet de quatre requêtes :

```
fetch_properties: ¦
    SELECT  prop_name, prop_id, prop_type
    FROM    prop_list

get_property_value: ¦
    SELECT  :prop_field
    FROM    prop_values
    WHERE   val_host_id = :host_id
      AND   val_prop_id = :prop_id

insert_property_value: ¦
    INSERT INTO prop_values
           ( :prop_field, val_host_id, val_prop_id )
    VALUES  ( :prop_value, :host_id,    :prop_id )

update_property_value: ¦
    UPDATE  prop_values
    SET     :prop_field = :prop_value
    WHERE   val_host_id = :host_id
      AND   val_prop_id = :prop_id
```

et de deux fonctions :

```
use constant { ID => 0, TYPE => 1 };

my $fetch_properties_qry =
   ➥$self->query("fetch_properties");
  $fetch_properties_qry->execute;

# l'instruction suivante construit un hash
# avec en clé les noms des propriétés et en
# valeur un arrayref qui contient l'ID et
# le type de la propriété
my %property = map { shift @$_, $_ }
   ➥@{ $fetch_properties_qry->
   ➥fetchall_arrayref };

sub get_property {
 my ($host_id, $prop_name) = @_;

 # détermination de l'ID et du type de la
```

```
 # propriété, et donc du champ à attaquer
 my $prop_id    = $property{$prop_name}[ID];
 my $prop_type  = $property{$prop_name}
  ➥[TYPE];
 my $prop_field = "val_t_$prop_type";

 my $get_prop_qry = $book->query(
   "get_property_value",
   replace => { prop_field => $prop_field }
 );

 $get_prop_qry->execute(
    host_id      => $host_id,
    prop_id      => $prop_id,
 );

 my ($prop_value) = $get_prop_sth->
     fetchrow_array;

 return $prop_value
}


sub set_property {
  my ($host_id, $prop_name, $prop_value)
  ➥ = @_;

  # détermination de l'ID et du type de la
  # propriété, et donc du champ à attaquer
  my $prop_id   = $property{$prop_name}[ID];
  my $prop_type   = $property{$prop_name}
  ➥[TYPE];
  my $prop_field = "val_t_$prop_type";

  # récupération de l'éventuelle ancienne
  # valeur pour savoir quelle requête
  # (INSERT ou UPDATE) utiliser
  my $old_value = get_property($host_id,
  ➥ $prop_name);
  my $set_prop_qry;
```

```
  if (defined $old_value) {
      if ($prop_value eq $old_value) {
      # petite optimisation, la valeur
      # existait  mais n'a pas changée
      # => rien à faire
        return
   }
   else {
        # la valeur existait et doit changer
        # => UPDATE
        $set_prop_sth = $self->query(
            "update_property_value",
            replace => { prop_field
          ➥          => $prop_field }
        );
      }
  }
  else {
      # il n'y avait pas de valeur => INSERT
      $set_prop_sth = $self->query(
        "insert_property_value",
        replace => { prop_field
       ➥     => $prop_field }
      );
  }

  $set_prop_sth->execute(
    host_id      => $host_id,
    prop_id      => $prop_id,
    prop_value   => $prop_value,
  );
}
```

Certes, le code peut sembler un peu complexe au premier abord, mais il est globalement court et facile à suivre. L'exemple présenté est une version réduite du système écrit par l'auteur qui utilise `Data::Phrasebook::SQL` en production, et qui permet de gérer l'ajout dynamique de propriétés et supporte des valeurs énumérées. L'ajout d'une nouvelle propriété se résume à une simple insertion dans

la table prop_list. L'ensemble étant de plus encapsulé de manière objet, les propriétés deviennent des attributs de l'objet, et le code final qui met à jour les propriétés ressemble à ceci :

```
my $host = Hebex::SysInfo::DWH::Host->new(
               host_name => $hostname
           );

# general operating system information
my $os_prop = os_properties($xmldoc);
$host->os_type( $os_prop->{type} );
$host->os_name( $os_prop->{name} );
$host->os_kernel( $os_prop->{kernel} );
$host->os_version( $os_prop->{version} );

# CPU information
$host->cpu_name(
  $xmldoc->findvalue("/conf/processor/name")
);
$host->cpu_vendor(
  $xmldoc->findvalue("/conf/processor/
  ➥ vendor")
);
$host->cpu_quantity(
  $xmldoc->findvalue("/conf/processor/
  ➥ quantity")
);
$host->cpu_frequency(
  $xmldoc->findvalue("/conf/processor/
  ➥ frequency")
);
$host->cpu_cache(
  $xmldoc->findvalue("/conf/processor/
  ➥ cache")
);

# network information
$host->default_gateway( $default_gateway );
```

L'auteur ne prétend pas que ceci est la meilleure manière de procéder, mais affirme en tout cas que cela marche bien... Et s'il est vrai que la complexité globale de l'applicatif n'est pas triviale, la complexité locale de chaque composant est très raisonnable, ce qui assure une maintenance plus facile du code. La séparation entre le code SQL et le code Perl est un point important, qui permet le partage du SQL entre plusieurs programmes indépendants, évitant les problèmes liés par exemple aux modifications de schéma.

ORM avec `DBIx::Class`

Perl dispose, depuis de nombreuses années, de frameworks ORM qui apparaissent au fur et à mesure de l'apparition de nouveaux besoins en capitalisant l'expérience acquise sur le framework précédent. Ainsi, le monde Perl a vu se succéder des modules comme MLDBM qui dès 1996 s'appuyait sur des bases BerkeleyDB pour stocker des structures Perl, puis Alzabo, Pixie et SPOPS au début des années 2000, suivis des excellents Tangram et `Class::DBI` jusqu'au milieu de cette même décennie.

ORM

Un ORM (*Object-Relational Mapping*) est un mécanisme permettant de transformer une base de données en une collection d'objets au niveau du langage de programmation. Il existe des ORM dans tous les langages.

Le point important d'un ORM est de pouvoir interroger une base sans jamais devoir écrire de requêtes SQL, le moteur de l'ORM se chargeant de les générer, les exécuter et de traduire les résultats sous forme objet, la seule que voit l'utilisateur.

L'intérêt, outre la manipulation d'objets rendue *a priori* plus facile – comparée à la manipulation d'enregistrements relationnels –, réside dans l'ajout d'un étage qui constitue une copie locale des données de la base, offrant une plus grande liberté pour travailler les relations entre ces données. Cela implique par contre de devoir synchroniser les objets en retour sur la base, pour sauver les modifications, et afin que cela fonctionne correctement, que le schéma des classes d'objets soit en concordance avec le schéma de la base.

Depuis 2005, l'ORM en Perl se réalise principalement au travers de `DBIx::Class`, qui est originellement né du besoin d'une refonte de `Class::DBI` pour le framework web Catalyst. `DBIx::Class` est en réalité un framework pour créer des ORM ; mais proposant par défaut une API (originellement celle de `Class::DBI`), il est donc suffisamment extensible pour que le besoin d'un nouveau modèle ne se fasse pas sentir pour le moment, du moins dans le cadre des bases SQL relationnelles.

`DBIx::Class` rend les opérations de manipulation et de synchronisation très simples. Il sait créer une base à partir d'un schéma objet, et inversement il sait construire un modèle objet à partir d'une base existante. Il propose également des outils facilitant la migration et la mise à jour de schémas.

Dans le schéma objet de `DBIx::Class`, les tables sont définies par des classes `Result` qui indiquent leurs colonnes et relations avec les autres tables.

Les requêtes pour interroger ces tables sont modélisées par des classes `ResultSet`. Un point intéressant à noter est que l'utilisation d'un `ResultSet` ne génère pas d'accès à la base. Seule la récupération des données va effectivement exécuter les requêtes SQL sous-jacentes et rapatrier les enre-

gistrements. Le but est de permettre de travailler le plus possible sur des ensembles virtuels, afin d'économiser les accès à la base.

Les données sont fournies sous la forme d'objets `DBIx::Class::Row`.

Créer un schéma `DBIx::Class`

Sans grande originalité, l'exemple typique du monde des ORM sera utilisé ici, soit une base de données référençant une collection musicale. Son schéma SQL est le suivant :

```
CREATE TABLE artist (
  artist_id INTEGER PRIMARY KEY,
  name      TEXT NOT NULL
);

CREATE TABLE album (
  album_id  INTEGER PRIMARY KEY,
  artist    INTEGER NOT NULL
    REFERENCES artist(artist_id),
  title     TEXT NOT NULL
);

CREATE TABLE track (
  track_id  INTEGER PRIMARY KEY,
  album     INTEGER NOT NULL
    REFERENCES album(album_id),
  title     TEXT NOT NULL
);
```

Il convient de définir les quelques classes mentionnées, en commençant par la classe mère `Schema` :

```
package MusicDB::Schema;
use parent qw< DBIx::Class::Schema >;
my $class = __PACKAGE__;

$class->load_namespaces();

1;
```

Puis une classe `Result` correspondant à une table `artist` :

```
package MusicDB::Schema::Result::Artist;
use parent qw< DBIx::Class::Core >;
my $class = __PACKAGE__;

$class->table("artist");
$class->add_columns(qw< artist_id name >);
$class->set_primary_key("artist_id");
$class->has_many(
  albums => "MusicDB::Schema::Result::Album"
  );

1;
```

Une classe pour la table `album` :

```
package MusicDB::Schema::Result::Album;
use parent qw< DBIx::Class::Core >;
my $class = __PACKAGE__;

$class->load_components("InflateColumn::
  ➥DateTime");
$class->table("album");
$class->add_columns(qw< album_id artist_id
  ➥title year >);
$class->set_primary_key("album_id");
$class->belongs_to(
   artist => "MusicDB::Schema::Artist",
   "artist_id");

1;
```

Et enfin une classe pour la table `track` :

```
package MusicDB::Schema::Result::Track;
use parent qw< DBIx::Class::Core >;
my $class = __PACKAGE__;

$class->table("track");
$class->add_columns(qw< track_id album
   ➥title >);
$class->set_primary_key("track_id");
$class->belongs_to(
  album => "MusicDB::Schema::Result::Album",
  "album_id");
```

En fait, ces classes sont terriblement sommaires, afin de rester lisibles, mais il manque ici toute la notion de typage, alors que `DBIx::Class` permet des définitions de types et de relations au moins aussi avancées qu'en SQL. Toutefois, cela ne présente que peu d'intérêt de devoir tout écrire à la main, surtout quand il existe déjà un module pouvant s'en charger, `DBIx::Class::Schema::Loader`.

Comme son nom l'indique, ce module charge le schéma depuis la base de données pour en extraire le plus d'informations possible et générer les classes Perl correspondantes. Il est fourni avec une commande `dbicdump` qui simplifie encore le travail. Son utilisation est la suivante :

```
dbicdump [-o <opt>=<value> ] <schema> <connect_info>
```

où `opt` est une option de `DBIx::Class::Schema::Loader::Base`, `schema` le nom de la classe de base qui définit le schéma et `connect_info` les informations de connexion, c'est-à-dire le DSN et les éventuels nom d'utilisateur et mot de passe.

Avec l'exemple précédent, la commande à exécuter est :

```
dbicdump -o dump_directory=lib MusicDB::Schema \
  'dbi:SQLite:dbname=music.db'
```

et elle génère dans le répertoire *lib/* les classes précédemment montrées, mais en bien mieux car celles-ci comprennent maintenant toutes les informations de types et de relations extraites du schéma SQL. Et de plus, les modules disposent d'une documentation minimale rappelant ces informations.

Utiliser un schéma DBIx::Class

Il faut avant tout charger le schéma et se connecter à la base :

```
use MusicDB;
my $music = MusicDB->connect("dbi:SQLite:
   ➥dbname=music.db");
```

Un album peut alors se récupérer directement par son identifiant, au travers du `ResultSet` correspondant de la table `album` :

```
my $album = $music->resultset("Album")->
   ➥find(32);
```

L'objet représente une copie en mémoire de l'enregistrement en base de l'album, directement modifiable :

```
$album->title("Fairy Dance");
```

Pour reporter les modifications en base, il faut invoquer la méthode `update()` :

```
$album->update;
```

Et inversement, pour jeter d'éventuelles modifications locales :

```
$album->discard_changes if $album->
   ➥is_changed;
```

Une recherche à partir du nom de l'artiste peut se réaliser au travers d'un `ResultSet` :

```
my $rs = $music->resultset("Album")->search(
  { artist => "Kokia" }
);
```

qui peut se parcourir comme un itérateur :

```
while (my $album = $rs->next) {
  say $album->title;
}
```

En contexte de liste, `search()` renvoie directement les objets résultat :

```
my @albums = $music->resultset("Album")->
  ➥search(
  { artist => "Aural Vampire" }
);
```

Stocker des objets avec KiokuDB

Si les ORM offrent une représentation objet d'une base de données, ils ne permettent toutefois pas directement de stocker des objets Perl natifs. Les premières tentatives en la matière ne sont pas récentes puisque c'était l'un des buts de Pixie dès 2000. Toutefois, la tâche n'était pas aisée à cause du modèle objet de base, très (trop) laxiste. Depuis, un changement est survenu avec `Moose`, et surtout le mécanisme sous-jacent, le protocole méta-objet (MOP), qui permet de construire des classes et des objets avec des fonctionnalités avancées, tout en conservant une capacité complète d'introspection.

C'est sur ces bases que Yuval Kogman s'est appuyé pour concevoir `KiokuDB`, un moteur de stockage de graphes

d'objets. En effet, dans un application un tant soit peu complexe, un objet est généralement relié à d'autres objets, ce qui forme un véritable graphe en mémoire. Sérialiser un objet nécessite donc souvent de pouvoir sérialiser le graphe complet. KiokuDB propose un framework transparent pour stocker les objets Perl. Il est spécialement conçu pour les objets s'appuyant sur Moose, mais supporte aussi les objets natifs « simples ».

Se connecter à une base KiokuDB

connect()

Son utilisation est remarquablement simple. Première étape, se connecter. Sans surprise, KiokuDB permet d'utiliser n'importe quelle base relationnelle classique au travers de DBI, par exemple, une petite base SQLite :

```
my $db = KiokuDB->connect(
  "dbi:SQLite:dbname=app_objects.db",
  create => 1,
);
```

et bien sûr, MySQL ou PostgreSQL :

```
my $db = KiokuDB->connect(
  "dbi:Pg:database=app_objects",
  user     => "app_user",
  password => "sekr3t",
  create   => 1,
);
```

Mais KiokuDB propose aussi des systèmes de stockage différents comme BerkeleyDB ou de simples fichiers :

```
# utilisation de BerkeleyDB
my $db = KiokuDB->connect(
  "bdb:dir=/path/to/storage",
```

```
  create => 1,
);

# utilisation de fichiers
my $db = KiokuDB->connect(
  "files:dir=/path/to/storage",
  serializer => "yaml", # Storable par déf.
  create => 1,
);
```

Ou encore les bases non-SQL comme CouchDB, MongoDB et Redis (voir page 243) :

```
my $db = KiokuDB->connect(
  "couchdb:uri=http://127.0.0.1:5984/
  ➥database",
  create => 1,
);
```

Il est aussi possible de fournir à `connect()` le chemin vers un fichier de configuration en YAML.

Stocker et récupérer des objets

new_scope()

store()

Une fois connecté, le stockage d'un objet est aussi simple que ces quelques lignes :

```
$db->new_scope();
my $uuid = $db->store($obj);
```

La méthode `new_scope()` crée une nouvelle portée dans laquelle réaliser les opérations sur un ensemble d'objets. C'est un moyen simple proposé par `KiokuDB` pour masquer

autant que possible le problème des références circulaires, qui pourraient sinon conduire à une non-libération de la mémoire. La méthode `store()` stocke l'objet et renvoie un identifiant unique, par défaut un UUID. Il est aussi possible de fournir directement l'identifiant :

```
$db->store($id => $obj);
```

Pour récupérer l'objet, il suffit donc de connaître son identifiant :

```
my $obj = $db->lookup($uuid);
```

Considérons un petit exemple. Voici une classe :

```
use MooseX::Declare;
class Node {
  has x => (isa => "Num", is => "rw",
  ➥ default => 0.0);
  has y => (isa => "Num", is => "rw",
  ➥ default => 0.0);
  has name => (isa => "Str", is => "rw",
  ➥ default => "");
  has peer => (isa => "Node", is => "rw",
  ➥ weak_ref => 1);
};
```

dont deux objets sont instanciés et liés entre eux, ce qui crée un graphe (certes simple, mais un graphe tout de même) :

```
my $node_A = Node->new(name => "A", x => 1,
   ➥y => 1);
my $node_B = Node->new(name => "B", x => 5,
   ➥y => 1);
$node_A->peer($node_B);
```

Leur stockage en base se fait naturellement :

```
$db->new_scope();
$db->store(node_A => $node_A, node_B =>
   ➥$node_B);
```

Le graphe comprenant ces deux objets est maintenant stocké, avec leur relation. Si plus tard, l'un d'eux est récupéré :

```
$db->new_scope();
my $node_A = $db->lookup("node_A");
```

le graphe complet est chargé, afin que l'attribut `peer` de cet objet ait bien quelque chose de correct au bout :

```
print Dumper( $node_A->peer );
# affiche :
# $VAR1 = bless( {
#                  'y' => 1,
#                  'name' => 'B',
#                  'x' => 5
#                }, 'Node' );
```

Administrer une base KiokuDB

Chose bien utile, une interface en ligne de commande est disponible via le module `KiokuDB::Cmd` qui installe la commande `kioku`. Celle-ci permet d'effectuer des opérations de maintenance sur une base `KiokuDB`, pour modifier des entrées (en les présentant dans un format lisible comme YAML ou JSON), vérifier et nettoyer les références internes.

Utiliser une base orientée paires de clé-valeur

Dans la catégorie des bases non-SQL, on trouve notamment Memcached, Tokyo Cabinet et Redis. Memcached est en réalité un cache mémoire partagé, donc volatile, mais très utile pour justement mettre en mémoire des données temporaires. Tokyo Cabinet et Redis sont par contre des bases persistantes, qui enregistrent les données.

Bases de données non-SQL

De nouveau types de bases de données sont apparus (ou réapparus) durant ces dernières années, dont la principale caractéristique est de n'être pas relationnelles et orientées enregistrements comme les bases SQL, mais plutôt orientées paires (de clé-valeur) ou orientées documents, et d'offrir des API natives plutôt qu'un langage de requête comme SQL. D'où leur regroupement derrière le nom générique de *bases non-SQL*, bien que cela soit un peu abusif puisque cela met sur le même plan des bases aux caractéristiques très différentes.

Elles ont des API logiquement assez proches. Ainsi pour Memcached :

```
use Cache::Memcached;

my $memd = Cache::Memcached->new(
    servers => [ "10.0.0.25:11211", "
        10.0.0.26:11211" ],
);

# ajout de valeurs
$memd->set(the_answer => 42);
```

```
# marche aussi avec des structures
$memd->set(result_set => { list => [ ... ]
    });

# récupération des valeurs
my $val = $memd->get("the_answer");
my $set = $memd->get("result_set");
```

Et avec Redis :

```
use Redis;

my $redis = Redis->new( server =>
   ➥"10.0.0.25:6379" );

# ajout de valeurs
$redis->set(the_answer => 42);

# récupération de valeurs
my $val = $memd->get("the_answer");
```

Le module Perl qui fournit le support Redis ne permet pas encore de stocker directement des références, mais il est toujours possible de les sérialiser manuellement.

Utiliser une base orientée documents

La notion de *document* doit ici se comprendre dans un sens assez générique. Il s'agit au final de structures stockées en l'état. La base la plus connue dans cette catégorie est CouchDB, qui utilise JSON (voir page 308) comme format. Plusieurs modules CPAN, aux API assez différentes, sont disponibles pour accéder à CouchDB, tels que `POE::Component::Client::CouchDB`, `CouchDB::Client` ou encore `AnyEvent::CouchDB`. Petit exemple avec ce dernier :

```
use AnyEvent::CouchDB;

my $couch = couch('http://localhost:5984/');
my $db    = $couch->db("app_objects");
my $host  = $db->open_doc("front01.domain
   ➥.net")->recv;
$host->{os}{arch} = "x86_64";
$db->save_doc($host)->recv;
```

L'un des intérêts de ces bases est de fournir un mécanisme pour stocker des objets sous une forme sérialisée et portable (comme JSON), ce qui permet de les interroger depuis n'importe quel langage de programmation, alors que les bases d'objets natifs comme `KiokuDB` ne peuvent bien évidemment fonctionner qu'avec des programmes Perl. CouchDB en particulier, par son stockage JSON, est même directement utilisable depuis du code JavaScript exécuté au sein d'une page web.

13

Dates et heures

La gestion du temps en informatique est un problème délicat car les concepts humains représentant le temps, les dates et les durées sont multiples, complexes et changeants. Pire encore, il s'agit d'un problème qui est même souvent d'ordre politique, d'ampleur internationale.

Cette description peut paraître exagérée, mais hélas la gestion *correcte* des dates et heures tient du casse-tête eschérien. Voici quelques petits exemples pour s'en convaincre.

Premièrement, le calcul d'une année bissextile : cela paraît une opération assez simple à effectuer. Il s'agit d'une année dont le millésime est divisible par 4, sauf celles divisibles par 100, mais en comprenant celles divisibles par 400. Ainsi, si 2000 et 2008 sont bissextiles, 1800, 1900 et 2010 ne le sont pas. (Et pour être complet, il faut signaler qu'il existe encore un décalage de trois jours sur un cycle de 10 000 ans à compenser.) Le calcul est *a priori* simple, et pourtant les consoles PlayStation 3 ont été affectées par un problème le 1er mars 2010 : celles-ci se croyaient dans une année bissextile, et donc un 29 février. D'où le dysfonctionnement de nombreux services. Le problème n'avait pas été détecté auparavant car la PS3 ayant été commercialisée en novembre 2006, 2010 était la première année paire non bissextile pour laquelle elle devait faire ce calcul (erroné. . .).

Microsoft avait affronté un problème similaire le 31 décembre 2008, lorsque ses lecteurs multimédia portables Zune (vendus seulement aux États-Unis) se sont tous bloqués, fuseau horaire après fuseau horaire, au fur et à mesure que ceux-ci passaient l'année. En effet, 2008 était une année avec une seconde intercalaire, ajoutée à la fin du 31 décembre, notée 23:59:60. Pas de chance pour Microsoft, le code de gestion des dates de Freescale ne gérait pas cette seconde, d'où le blocage général.

Et à propos de fuseaux horaires, sur combien de fuseaux s'étend la Russie ? Neuf depuis le 28 mars 2010, date à laquelle, sur décision du Président Dmitri Medvedev, deux des onze fuseaux jusqu'alors utilisés ont été abandonnés (UTC+4 et UTC+12).

Enfin, le passage de l'heure d'hiver à l'heure d'été ne s'effectue pas au même moment dans tous les pays. Pire, pour certains États fédéraux comme les États-Unis d'Amérique, la décision se prend au niveau local et non fédéral, d'où certains États qui, bien que situés dans un même fuseau horaire, ne sont pas à la même heure.

La base tz

Les États publient bien évidemment les informations liées par exemple aux changements d'horaire, mais il n'existe pas d'organisme international qui regrouperait toutes ces données pour les compiler. Fort heureusement, l'apparition des réseaux informatiques a permis la mise en place d'une collaboration internationale pour la constitution d'une base de données prévue pour l'usage informatique, compilant justement les informations publiées par les États et organismes en charge de la gestion du temps. Cette base est nommée base `tz`, pour *timezone*, ou encore base Olson, d'après Arthur David Olson qui a

initié le projet en 1986 et a publié dans le domaine public le code source pour utiliser les données de cette base, aujourd'hui principalement maintenue par Paul Eggert.

Un aspect important de la base Olson est qu'elle définit les « zones de temps » (*time zones*) qui donnent leur nom à cette base, mais qui sont très différentes des fuseaux horaires. En effet, ces zones sont définies comme des régions géographiques où les horloges locales sont en accord depuis 1970, une zone pouvant couvrir un État ou une partie d'un État. Il faut noter que cette définition, contrairement à celle des fuseaux horaires, n'est pas normative.

Ces zones sont nommées suivant le schéma *aire/zone*, par exemple « Europe/Paris », ou « America/New_York », tous les noms de lieux étant en anglais. L'aire est le nom d'un continent (Asia, Africa) ou d'un océan (Atlantic, Pacific). La plupart des zones correspondent par construction à un pays, sauf pour certains pays, en particulier les plus étendus (Russie, États-Unis, Canada, Chine, Brésil) qui sont divisés en plusieurs zones. Dans tous les cas, les zones ont pour nom celui de la ville la plus peuplée ou la plus représentative de la région.

Utiliser le module `Date::Parse`

Une date peut être écrite sous des formats très divers : ISO-8601 dans le meilleur des cas, `ctime(3)` qui se rencontre dans bien des fichiers de journalisation, *timestamp* Unix dans certains cas, ou encore des formats personnalisés (comme celui d'Apache). Dans certains cas, il peut manquer des informations. Un cas typique est celui des journaux de serveurs de mails (Sendmail, Postfix) dans lesquels l'année et le fuseau horaire sont souvent absents.

Il est bien sûr possible de vouloir gérer cela « à la main », mais pourquoi se compliquer la vie et perdre du temps à découvrir tous les problèmes qu'il faut résoudre quand un module de la qualité de `Date::Parse` est disponible sur CPAN ? Il est en effet capable d'analyser un grand nombre de formats différents, et supporte même plusieurs langues.

Son interface est constituée des deux fonctions `str2time()` et `strptime()` qui toutes deux acceptent une chaîne en argument, ainsi qu'un nom de zone en second argument optionnel.

Lire une date avec `Date::Parse`

`str2time(..)`

`str2time()` donne comme résultat le timestamp Unix correspondant :

```
use Date::Parse;

# format Apache
my $date = "21/Jun/2010:23:09:17 +0200";
my $time = str2time($date);
print $time;     # "1277154557"
```

Interpréter une date avec `Date::Parse`

`strptime(..)`

`strptime()` renvoie une liste de valeurs correspondant aux différentes composantes de la date :

```
use Date::Parse;

# format Apache
my $date = "21/Jun/2010:23:09:17 +0200";
my ($ss, $mm, $hh, $day, $month, $year,
   ➥$zone) = strptime($date);
$month += 1;
$year  += 1900;
print "$day/$month/$year - $hh:$mm:$ss
   ➥($zone)";
# affiche "21/6/2010 - 23:09:17 (7200)"
```

Info

Ici, `$zone` est le décalage en secondes de la zone par rapport à GMT. Il faut noter que `strptime()` suit les mêmes conventions que `localtime()` et les autres fonctions internes de gestion du temps, et que le mois est donc de 0 à 11, et l'année est retranchée de 1900. Il est donc nécessaire de corriger ces deux valeurs pour toute utilisation où les intervalles normaux sont attendus, que ce soit pour un affichage destiné à un humain ou comme argument à un module comme `DateTime`.

Changer la langue avec **Date**::**Language**

```
Date::Language->new("French");
```

Par défaut, `Date::Parse` n'analyse que les dates en anglais, mais supporte près d'une trentaine de langues différentes par le biais du module `Date::Language`, qui propose une API proche si ce n'est qu'il faut créer un objet correspondant à la langue d'analyse :

```
use Date::Language;

my $date = "21 juin 2010 04:15:31";

# création de l'objet Date::Language
# pour analyser en français
my $lang = Date::Language->new("French");

my $time = $lang->str2time($date);
print "$time";
# affiche "1277086531"

my ($ss, $mm, $hh, $day, $month, $year,
   ➥$zone) = $lang->strptime($date);
$month += 1;
$year  += 1900;
print "$day/$month/$year - $hh:$mm:$ss
   ➥($zone)";
# affiche "21/6/2010 - 04:15:31 ()"
```

La zone est ici logiquement vide puisque aucune n'était précisée dans la date fournie. Il faut d'ailleurs noter que `strptime()` ne fournit des valeurs que s'il a pu les déterminer, et renvoie un tableau vide en cas d'erreur.

Gérer les intervalles de temps avec `Time::Duration`

Un autre besoin très courant est d'afficher sous forme compréhensible le délai avant la fin d'une tâche, ou la durée entre deux dates. Les conversions sont bien évidemment simples à écrire, mais le module `Time::Duration`, outre cette petite factorisation, apporte un plus non négligeable sous la forme d'une approximation, d'une réduction de l'information afin de la rendre plus lisible.

Interpréter une durée avec Time::Duration

```
duration(..)
```

La fonction de base du module `Time::Duration` est `duration()`, qui traduit une durée compréhensible par un être humain :

```
print duration(3820);
# affiche "1 hour and 4 minutes"
```

Astuce

Ce n'est pas immédiatement visible, mais la fonction a ici arrondi le résultat. La valeur exacte peut être obtenue ainsi :

```
print duration_exact(3820);
# affiche "1 hour, 3 minutes,
   ➥and 40 seconds"
```

Par défaut, `duration()` réduit le nombre d'unités à afficher à deux, car un être humain s'inquiète par exemple rarement d'avoir d'une précision de l'ordre de la seconde si l'attente affichée est de l'ordre de l'heure. Le nombre d'unités affichées peut bien sûr se contrôler par un second argument :

```
print duration(38487592, 3);
# affiche "1 year, 80 days, and 11 hours"
# ce qui reste plus lisible que la valeur
# exacte "1 year, 80 days, 10 hours,
# 59 minutes, and 52 seconds"
```

Obtenir la durée à partir de maintenant

```
ago(..) later(..)
```

`Time::Duration` propose une série de fonctions qui permettent d'en exprimer un peu plus :

```
print ago(120);          # "2 minutes ago"
print later(3750, 1);    # "1 hour later"
```

Astuce

L'exemple précédent n'est certes pas transcendant, mais ces fonctions ont l'avantage d'être un peu plus malignes :

```
print ago(0);         # "right now"
print ago(-130);
# "2 minutes and 10 seconds from now"

print later(0);       # "right then"
print later(-2460); # "41 minutes earlier"
```

Ainsi, quand `ago()` reçoit une valeur négative, elle est passée à `from_now()`, et inversement ; de même pour les fonctions `later()` et `earlier()`.

Réduire l'affichage

```
concise(..)
```

Une fonction de `Time::Duration`, `concise()`, permet de réduire les chaînes produites par les autres fonctions :

```
print ago(5938);
# affiche "1 hour and 39 minutes ago"

print concise(duration(5938));
# "1h39m ago"
```

Changer la langue avec Time::Duration::fr

```
use Time::Duration::fr
```

L'auteur de `Time::Duration` a pensé à l'internationalisation, et a rendu possible l'écriture de sous-classes permettant la localisation. Comme il n'existait pas de version française, l'un des auteurs de ce livre s'en est chargé, ce qui ne lui a demandé que quelques heures durant un long voyage en train... Le module `Time::Duration::fr` est disponible sur CPAN :

```
use Time::Duration::fr;

print duration(243550, 3);
# "2 jours, 19 heures, et 39 minutes"
print ago(243550);
# "il y a 2 jours et 20 heures"
print earlier(243550, 1);
# "3 jours plus tôt"
```

Info

Voici quelques exemples plus concrets :

```
my $file = "data-dumper-ex2";
my $age = $^T - (stat($file))[9];
print "Le fichier a été changé ",
   ➥ago($age), $/;
```

```
# "Le fichier a été changé il y a 3 jours
#  et 47 minutes"

print "Expiration : ", from_now(7390), $/;
# "Expiration  : dans 2 heures et 3 minutes"

print "Cette action sera exécutée ",
      later(245), " que la précédente.";
# "Cette action sera exécutée 4 minutes et
#   5 secondes plus tard que la précédente."
```

Utiliser les modules `DateTime`

`DateTime::Duration`
`DateTime::Format`

La gestion des dates, heures, durées et autres questions liées est complexe. `DateTime` est un ensemble de modules CPAN stables et matures qui allient facilité d'utilisation pour les cas simples et fonctionnalités puissantes pour résoudre les problèmes complexes :

- `DateTime` : le module principal est `DateTime`, qui permet de créer des objets correspondant à des moments précis dans le temps.
- `DateTime::Duration` : il permet de représenter une durée, un intervalle entre deux dates.
- `DateTime::Format` : accompagné de ses modules dérivés, ils permettent de transformer des données (généralement du texte) en dates, et surtout de produire des représentations (souvent textuelles) de dates et durées.
- `DateTime::Event` : les modules `DateTime::Event::*` contiennent des listes d'événements connus, pouvant être utilisés facilement.

Construire une instance DateTime arbitraire

```
DateTime->new(. . .)
```

DateTime propose principalement deux constructeurs pour représenter un moment dans le temps : new() et now().

```
use DateTime;
my $christmas = DateTime->new( year => 2010,
                               month  => 12,
                               day    => 25,
                             );
```

La variable $christmas est maintenant un objet DateTime, qui représente le 25 décembre 2010. On peut s'en persuader en l'affichant :

```
print $christmas;
# affiche 2010-12-25T00:00:00
```

Le seul argument obligatoire à new() est year, les autres prendront des valeurs par défaut.

Il est possible d'être plus précis et de rajouter des paramètres à new() :

```
use DateTime;
my $christmas = DateTime->new(
    year => 2010,
    month  => 12,
    day    => 25,
    hour   => 14,
    minute => 12,
    second => 42,
    nanosecond => 600000000,
    );
```

Ici, plus d'arguments ont été passés, permettant d'indiquer presque parfaitement un moment précis dans le temps. Il manque cependant une notion, celle du fuseau horaire.

Choisir un fuseau horaire

```
time_zone => 'Europe/Paris'
```

Par défaut, lorsque le fuseau horaire n'est pas précisé, `DateTime` utilise un fuseau horaire spécial, dit « flottant » : lors des calculs et manipulations de temps, les changement de fuseau horaire ne seront pas pris en compte. Pour des situations simples, c'est suffisant, mais il est également possible de spécifier le fuseau horaire lors de la création d'un objet `DateTime` :

```
use DateTime;
my $christmas_in_london = DateTime->new(
      year   => 2010,
      month  => 12,
      day    => 25,
      time_zone => 'Europe/London'
);
my $christmas_in_paris = DateTime->new(
    year   => 2010,
    month  => 12,
    day    => 25,
    time_zone => 'Europe/Paris'
);
```

Obtenir l'instant présent

```
DateTime->now()
```

Il est facile de créer un objet `DateTime` correspondant à l'instant présent, en utilisant le constructeur `now()`, qui comme `new()` peut prendre comme paramètre `time_zone`.

```
use DateTime;
my $now = DateTime->now();
sleep 5;
my $five_seconds_later = DateTime->now();
```

Obtenir la date du jour

```
DateTime->today()
```

Il existe également un raccourci pour obtenir un objet date du jour, en utilisant `today()`.

```
use DateTime;
my $today = DateTime->today();
```

Obtenir l'année

```
$dt->year()
$dt->set_year(..)
```

Retourne l'année de l'objet. `$dt->set_year()` permet de mettre à jour la valeur.

Info

Les objets `DateTime` ont beaucoup de méthodes pour récupérer et modifier leur attributs. Il est inutile de les détailler toutes ici, seules les plus importantes sont présentées.

Obtenir le mois

```
$dt->month()
$dt->set_month(..)
```

Retourne le mois comme entier de 1 à 12. `$dt->set_month()` permet de mettre à jour la valeur.

Obtenir le nom du mois

```
$dt->month_name()
```

Retourne le nom du mois, conformément à la locale.

Obtenir le jour du mois

```
$dt->day_of_month()
```

Retourne le jour du mois, de 1 à 31. `$dt->set_day()` permet de mettre à jour la valeur.

Obtenir le jour de la semaine

```
$dt->day_of_week()
```

Retourne le jour de la semaine, de 1 à 7.

Obtenir le nom du jour

```
$dt->day_name()
```

Retourne le nom du jour de la semaine, conformément à la locale.

Obtenir l'heure

```
$dt->hour()
$dt->set_hour()
```

Retourne l'heure, de 0 à 23. `$dt->set_hour()` permet de mettre à jour la valeur.

Obtenir les minutes

```
$dt->minute()
$dt->set_minute()
```

Retourne les minutes, de 0 à 59. `$dt->set_minute()` permet de mettre à jour la valeur.

Obtenir les secondes

```
$dt->second()
$dt->set_second()
```

Retourne les secondes, de 0 à 61. Les valeurs 60 et 61 sont utilisées pour les secondes intercalaires. `$dt->set_second()` permet de mettre à jour la valeur.

Les secondes intercalaires

Les secondes intercalaires sont des secondes. Selon Wikipedia, *une seconde intercalaire, également appelée saut de seconde ou seconde additionnelle, est un ajustement d'une seconde du Temps universel coordonné (UTC). Et ce, afin qu'il reste assez proche du Temps universel (UT) défini quant à lui par l'orientation de la Terre par rapport aux étoiles.*

Il s'agit d'un système, qui date de 1972, et qui met en place des moments où une seconde est retranchée ou ajoutée. Cela permet de garder le Temps universel coordonné à moins de 0,9 seconde du Temps universel (UT). Les besoin d'ajout ou de retranchement de ces secondes intercalaires ne peuvent pas être prévus, le mouvement de la Terre n'étant pas prévisible avec une très grande précision.

C'est pourquoi il est possible d'avoir une date avec un nombre de secondes supérieur à 59, jusqu'à 61.

Obtenir les nanosecondes

```
$dt->nanosecond()
$dt->set_nanosecond()
```

Retourne les nanosecondes. Par exemple, 500 000 000 est une demi-seconde. `$dt->set_nanosecond()` permet de mettre à jour la valeur.

Obtenir des durées de temps

DateTime::Duration->new()

Un objet de type `DateTime` représente un instant donné. Pour pouvoir manipuler des temps et dates plus facilement, il est très utile de disposer du concept de durée. La classe `DateTime::Duration` implémente ce concept. Les objets qui en sont instanciées représentent des durées arbitraires, qui peuvent être modifiées et utilisées pour modifier ou créer de nouvelles dates.

Un objet `DateTime::Duration` peut s'obtenir à partir de rien, en utilisant son constructeur `new()` :

```
my $duration = DateTime::Duration->new(
   ➥years => 1,
   ➥months => 6,
);
# $duration représente un an et 6 mois
```

Les paramètres du constructeur sont donnés dans le tableau suivant.

Tableau 13.1: Constructeur de DateTime::Duration

Paramètre	Description
years	le nombre d'années
months	le nombre de mois
weeks	le nombre de semaines
days	le nombre de semaines
hours	le nombre d'heures
minutes	le nombre de minutes
seconds	le nombre de secondes
nanoseconds	le nombre de nanosecondes

Astuce

Il est également possible de créer une durée directement à partir de deux dates :

```
my $dt1 = DateTime->new( year   => 2011,
                         month  => 12,
                         day    => 25,
                       );
my $dt2 = DateTime->new( year   => 2010,
                         month  => 12,
                         day    => 25,
                       );
my $duration = $dt1->subtract_datetime($dt2)
    ;
# $duration représente la durée entre
# les deux dates, donc un an.
```

Décaler une date dans le futur

`$dt->add(..)`

La méthode `add()` permet de décaler un objet `DateTime` dans le futur.

```
# date représentant Noël 2010
my $dt = DateTime->new( year   => 2010,
                        month  => 12,
                        day    => 25,
                      );
$dt->add( year => 1 );
# $dt représente maintenant Noël 2011

$dt->add( days => 1 );
# $dt représente maintenant
# le 26 décembre 2011
```

Les paramètres sont les mêmes que ceux du constructeur de la classe `DateTime::Duration` : donc `years`, `months`, `weeks`, `days`, `hours`, `minutes`, `seconds`, `nanoseconds`.

Ajouter une durée

```
$dt->add_duration(..)
```

Cette méthode est similaire à `add()`, cependant elle prend en argument une durée, c'est-à-dire un objet `DateTime::Duration`.

```
use DateTime;
use DateTime::Duration;
my $duration = DateTime::Duration->new(
   ➥days => 15 );
my $dt = DateTime->now();
$dt->add_duration($duration);
# $dt est maintenant 15 jours
# dans le futur
```

Attention

Il ne faut pas utiliser `add()` ou `add_duration()`avec des valeurs négatives pour décaler un objet `DateTime` dans le passé. Pour cela, utiliser la méthode `subtract()`.

Décaler une date dans le passé

```
$dt->subtract(..)
```

La méthode `subtract()` permet de décaler un objet `DateTime` dans le passé.

```
# date représentant noël 2010
my $dt = DateTime->new( year   => 2010,
                        month  => 12,
                        day    => 25,
                      );

$dt->subtract( year => 1 );
# $dt représente maintenant noël 2009

$dt->subtract( days => 1 );
# $dt représente maintenant
# le 24 décembre 2009
```

Cette méthode est très similaire à `add()`, et accepte les même paramètres. La différence est que les valeurs sont retranchées, et non ajoutées.

Soustraire une durée

`$dt->subtract_duration(..)`

Cette méthode est similaire à `subtract()`, cependant elle prend en paramètre une durée, c'est-à-dire un objet `DateTime::Duration`.

```
use DateTime;
use DateTime::Duration;
my $duration = DateTime::Duration->new(
   ➥ days => 15 );
my $dt = DateTime->now();
$dt->subtract_duration($duration);
# $dt est maintenant 15 jours dans le passé
```

Calculer un intervalle de temps

```
$dt->subtract_datetime(..)
delta_md()
delta_days()
delta_ms()
```

Cette méthode permet de calculer l'intervalle de temps entre deux objets `DateTime`. Elle retourne un objet `DateTime::Duration`, relativement à l'objet sur lequel elle est appliquée :

```
# date représentant noël 2010
my $dt_10 = DateTime->new( year   => 2010,
                           month  => 12,
                           day    => 25,
                         );
# date représentant noël 2009
my $dt_09 = DateTime->new( year   => 2009,
                           month  => 12,
                           day    => 25,
                         );
my $duration = $dt_10->subtract_datetime(
   ➥ $dt_09 );
# $duration représente 1 an
```

Autre possibilité, les méthodes `delta_*`. `delta_md` renvoie une `DateTime::Duration` qui représente une portion de la différence entre deux objets `DateTime`.

```
$dt->delta_md($datetime);
$dt->delta_days($datetime);
$dt->delta_ms($datetime);
```

- `delta_md()` : cette méthode renvoie une durée exprimée en mois et en jours.

- `delta_days()` : cette méthode renvoie une durée exprimée uniquement en jours.
- `delta_ms()` : cette méthode renvoie une durée exprimée uniquement en minutes et secondes.

Attention

Ces méthodes renvoient toujours une durée positive ou nulle.

Les modules formateurs

Il existe beaucoup de modules dérivés de `DateTime::Format`. Ils sont désignés sous le terme `DateTime::Format::*`.

Par abus de langage, un « formateur » (ou *formatter* en anglais) désigne un de ces modules (par exemple `DateTime::Format::Pg`), ou bien une instance de ce module.

Un formateur est un module qui va permettre tout d'abord d'interpréter une chaîne de texte, et de construire un objet `DateTime`, et également d'utiliser un objet `DateTime` pour en générer une représentation textuelle.

Générer une représentation textuelle d'une date

```
format_datetime(..)
```

Tous les formateurs ont un point commun : ils implémentent une méthode `format_datetime`, qui permet de générer la représentation texte d'un objet `DateTime`. Il suffit de l'utiliser et d'afficher la chaîne de caractères :

```
use DateTime;
use DateTime::Format::Human;
my $formatter = DateTime::Format::Human->
   ➥new();

my $spoken_time = $formatter->
   ➥format_datetime(
                        DateTime->now()
                    );

say "The time is now $spoken_time";
```

Cet exemple affichera sur la console :

```
The time is now a little after quarter past eight
in the evening
```

Astuce

Il est possible d'attacher directement un formateur à un objet `DateTime`. Ainsi, dès que l'objet en question est utilisé dans un contexte de chaîne[1], le formateur sera utilisé de manière transparente. Un exemple est plus parlant :

```
use DateTime;
use DateTime::Format::Strptime
my $formatter = DateTime::Format::Strptime->
   ➥new(
           pattern      => '%T'
       );
my $now = DateTime->now( formatter =>
   ➥$formatter );
say "Il est maintenant : $now";
```

Ce code affichera quelque chose comme :

```
Il est maintenant : 18:15:35
```

1. Voir le contexte de chaîne au Chapitre *Éléments du langage* page 22.

La majorité des formateurs prennent des arguments, qui permettent de paramétrer le comportement de la représentation texte générée : n'afficher qu'une partie de l'information (par exemple juste les heures), spécifier le langage, l'ordre des informations, etc.

Interpréter une date

parse_datetime(..)

La plupart des formateurs permettent d'interpréter une chaîne de caractères, et de renvoyer un objet `DateTime` correspondant. Pour cela, il faut généralement leur donner quelques arguments pour les aider à reconnaître la date correctement. Voici un exemple simple avec `DateTime::Format::Strptime` :

```
use DateTime::Format::Strptime;
my $formatter = new DateTime::Format::
    Strptime(
    pattern      => '%T',
    locale       => 'en_FR',
    time_zone    => 'Europe/Paris',
);

my $dt = $formatter->parse_datetime(
   ➥'22:14:37');
```

Attention

Certains formateurs n'implémentent pas l'interprétation de dates, mais juste la génération de texte. Il faut se référer à leur documentation (par exemple sur CPAN) pour vérifier leur fonctionnalités.

Il serait bien trop long d'examiner ici les quatre-vingts modules de formatage `DateTime::Format::*`. Voici une liste des modules les plus intéressants et utiles, à consulter sur CPAN :

- `DateTime::Format::Natural` : probablement le plus polyvalent et utile, il permet d'interpréter et générer des dates qui ont un format courant, que l'on retrouve dans les écrits. Ce module est paramétrable très finement, pour s'adapter à presque toutes les manières qu'ont les hommes d'écrire une date.
- `DateTime::Format::Strptime` : comme vu précédemment, permet de manipuler les représentations de dates aux formats *strp* et *strf*, très utilisés en informatique.
- `DateTime::Format::DBI` : permet de travailler avec les représentations de date dans les bases de données. Voir aussi Oracle, `DateTime::Format::SQLite`, `DateTime::Format::Oracle`, etc.
- `DateTime::Format::Human` : comme vu précédemment, permet de générer la date dans un format intelligible pour un humain. `DateTime::Format::Human::Duration` permet de générer et interpréter des durées.
- `DateTime::Format::Builder` : permet de créer ses propres formateurs spécifiques, à partir de différentes règles.

14

XML

Cette partie est consacrée à la manipulation des formats structurés, c'est-à-dire des données qui suivent une structure plus ou moins bien définie. Étant pour la plupart en format texte, donc faciles à lire, ils semblent simples à gérer par programmation. Mais cette apparente simplicité est souvent trompeuse en raison d'une part de la complexité intrinsèque des formats, et d'autre part de la grande variabilité de conformité des données que le développeur, ou son programme, est amené à rencontrer dans la vie réelle. Même s'il semble souvent plus rapide, ou facile, de sortir l'arme des expressions régulières pour analyser le format, cela s'avère dans quasiment tous les cas une *mauvaise idée*.

La bonne manière de faire est d'utiliser le ou les modules appropriés[1], auxquels il faut déléguer le rôle de correctement lire et écrire les données en respectant leur grammaire. L'abstraction que ces modules apportent permet aussi, bien souvent, de simplifier le code en aval, sans parler du fait que ceux qui seront présentés ici sont intensivement testés et utilisés depuis des années par un grand nombre de personnes.

1. Ce rappel peut sembler superflu, mais les auteurs ont déjà croisé plus d'un analyseur HTML en expressions régulières, et voudraient ne pas en voir davantage dans la nature…

Est-il encore besoin de présenter XML ? Rappelons que plus qu'un simple format, il s'agit d'une grammaire qui permet de définir des formats, appelés *applications* en terminologie XML. XHTML (la reformulation de HTML conforme XML), SOAP, RSS ou encore OpenDocument (le format des fichiers OpenOffice.org) sont des applications XML, mais qui correspondent clairement à des formats bien différents.

Perl dispose de très nombreux modules liés au traitement du XML. Trop nombreux même, mais c'est le cas dans tous les langages qui ont des bibliothèques pour manipuler le XML depuis la première publication de cette norme en 1998. Modules d'analyse, de génération, d'API standard ou non d'accès au document (DOM, SAX, XPath), ou de transformation (XSLT), Perl dispose de modules pour à peu près tous les cas et avec la plupart, si ce n'est toutes, les bibliothèques externes qui forment les standards actuels. Nous n'allons vraiment examiner que deux modules, car ils suffisent à combler la majeure partie des besoins courants.

DOM et SAX

Petit rappel pour ceux qui ne connaîtraient pas certains des nombreux noms bizarres dont regorgent les spécifications XML. *DOM* (*Document Object Model*) est un modèle où le document XML est présenté sous la forme d'un arbre fini, manipulable au travers de fonctions normalisées. DOM est un standard édicté par le W3C, afin de définir une API universelle, où les fonctions ont les mêmes noms et mêmes arguments quel que soit le langage de programmation.

L'ensemble de la structure du document étant en mémoire, il est très facile de parcourir les branches de l'arbre, récupérer

et modifier des informations, et même changer la structure du document en ajoutant ou supprimant des branches.

Ce modèle est pratique pour les documents de faible taille, mais sa consommation mémoire le rend rapidement inutilisable pour les documents de taille plus importante. La limite exacte dépend de la consommation mémoire admise, mais au delà de la centaine de méga-octets, c'est généralement un gaspillage de ressources.

SAX (*Simple API for XML*) fonctionne au contraire sur un modèle événementiel, invoquant des méthodes définies par le programmeur au fur et à mesure de l'analyse. Seul le nœud courant est visible, et il n'est pas possible d'accéder à une autre partie du document.

Très rapide d'exécution, cette méthode consomme aussi peu de ressources puisque seule une petite partie du contenu XML est chargée en mémoire à chaque instant. C'est donc une technique conçue pour le traitement des documents de grande taille, qui ne peuvent tenir en mémoire. Elle est aussi adaptée au traitement des flux XML continus.

Son inconvénient est logiquement l'absence de contexte, qui impose au programme de reconstituer celui dont il a besoin en stockant en parallèle les informations nécessaires.

Charger un document XML avec `XML::LibXML`

Derrière un nom assez peu parlant se cache le couteau suisse du XML en Perl, car `XML::LibXML` est le *binding* de la `libxml2`, la bibliothèque maintenue par la Fondation Gnome et le W3C. Il propose, outre l'analyse de XML

avec support des espaces de noms, la validation contre des DTD, l'analyse de document HTML/4, le support de XPath, XPointer, XInclude et offre les API DOM et SAX.

Malgré toutes ces fonctionnalités supportées, le module reste tout de même simple à utiliser :

```
use XML::LibXML;

my $parser = XML::LibXML->new;
my $xmldoc = $parser->parse_file($path);
```

Ces quelques lignes suffisent pour analyser un document XML dont on a donné le chemin (`$path`) et obtenir l'arbre DOM correspondant dans `$xmldoc`. L'analyseur accepte bien sûr de nombreuses options, par exemple pour la gestion des erreurs voire l'activation de la validation du document, l'expansion des entités et DTD, le support XInclude, etc.

`XML::LibXML` accepte comme source le chemin d'un fichier (`parse_file()`), un descripteur de fichier (`parse_fh()`) ou encore une chaîne de caractères (`parse_string()`).

L'analyse d'une page web n'est guère plus complexe :

```
use LWP::Simple;
use XML::LibXML;

my $parser  = XML::LibXML->new;
my $html    = get("http://mongueurs.net/");
my $htmldoc = $parser->parse_html_string(
   ➥$html);
```

À noter que `XML::LibXML` est bien moins permissif que `HTML::Parser` sur les documents HTML qu'il accepte de considérer comme valides, mais il peut s'avérer utile sur des documents de type XHTML.

Il faut remarquer que `XML::LibXML` propose depuis peu une nouvelle API, un peu plus souple, sous la forme des méthodes `load_xml()` et `load_html()` qui permettent de spécifier la source de données sous la forme d'une option, `location` pour lire depuis un fichier local ou depuis une URL :

```
my $xmldoc = $parser->load_xml(location =>
  ➥$location);
```

`string` pour lire depuis une chaîne :

```
my $xmldoc = $parser->load_xml(string =>
  ➥$xml_content);
```

ou encore `IO` pour lire depuis un descripteur de fichier :

```
my $xmldoc = $parser->load_xml(IO => $fh);
```

Ces méthodes peuvent même être invoquées sous forme de méthodes de classe, le module se chargeant de créer l'analyseur, avec les options par défaut.

Dans tous les cas, l'objet renvoyé représente l'arbre DOM du document XML fourni en entrée.

Parcourir un arbre DOM

Pour regarder cela plus en détail, prenons le fichier XML suivant comme exemple, qui décrit des informations sur une machine :

```
<?xml version="1.0"?>
 <host>
   <name>www01.dcf.domain.net</name>
   <network>
     <interface name="eth0">
     <address type="ipv4" prefix="24">10.0.0.132</address>
       <address type="ipv6" prefix="64"
       >2a01:e35:2f26:dde0:213:51af:fe7c:b4d8</address>
```

```
    </interface>
  </network>
  <os>
    <type>Mac OS X</type>
    <version>10.6.4</version>
   <kernel>Darwin Kernel Version 10.4.0: Fri Apr 23 18:28:53
    PDT 2010; root:xnu-1504.7.4~1/RELEASE_I386</kernel>
    <arch>i486</arch>
  </os>
  <uptime>5895074.06</uptime>
  <!-- ... -->
</host>
```

Le constructeur fournit un objet représentant l'arbre DOM :

```
my $dom = $parser->parse_file(
  ➥"www01.dcf.domain.net.xml");
```

Une première manière de parcourir cet arbre est d'utiliser l'API DOM :

```
my $root = $dom->documentElement();
my ($node) = $root->getChildrenByTagName(
  ➥"name");
print $node->nodeName(), " = ",
  ➥$node->textContent(), "\n";
```

Même si les noms des méthodes sont très explicites, ce premier bout de code montre tout ce qu'il faut écrire pour simplement accéder au premier fils du nœud racine. Pour afficher l'adresse IPv4 de l'interface réseau, en parcourant scrupuleusement l'arbre, il faut dérouler encore plus de code :

```
my ($network)   =
  ➥$root->getChildrenByTagName("network");
my ($interface) =
  ➥$network->getChildrenByTagName("
    interface");
my (@nodes)     =
```

```
    ➥$interface->getChildrenByTagName("
      address");

for my $node (@nodes) {
  next unless $node->nodeType ==
    ➥XML_ELEMENT_NODE;

  if ($node->getAttribute("type") eq "ipv4")
    {print $node->textContent, "\n";}
}
```

En réalité, il y a moyen de raccourcir un peu et d'aller directement aux nœuds `address` :

```
my @nodes = $root->getElementsByTagName(
   ➥"address");

for my $node (@nodes) {
  if ($node->getAttribute("type") eq "ipv4")
    {print $node->textContent, "\n";}
}
```

mais cela ne fonctionne bien que si le document a des éléments dont les noms sont tous non ambigus, mais en général alors peu digestes comme `netInterfaceAddress`. Si le nom est trop générique, comme `type` ou `name`, `getElementsByTagName()` est inutilisable.

Cet exemple est un peu injuste car, en ne montrant que la lecture de valeurs, il met en exergue la lourdeur de cette API, qui n'est pas à imputer à `XML::LibXML`, mais bien au standard DOM Level 3 qu'il respecte scrupuleusement. Mais pour être honnête, il faut signaler qu'elle est extrêmement complète, et permet de construire un arbre à partir de zéro, d'ajouter de nouveaux nœuds, qui peuvent recevoir attributs, textes et nœuds fils.

Toutefois, il faut bien reconnaître que pour le cas somme toute très courant de la simple exploitation d'un document, DOM n'est pas ce qu'il y a de plus simple. C'est là

où `XML::LibXML` apporte ce qui fait son plus par rapport à bien d'autres modules XML, le support de XPath.

Utiliser XPath

XPath est aussi une norme édictée par le W3C, qui permet de naviguer et d'effectuer des recherches au sein d'un arbre DOM au moyen d'un chemin inspiré, dans sa forme la plus simple, de la syntaxe des chemins Unix : les noms sont séparés par des slashs, `'.'` représente le nœud courant et `'..'` le nœud parent.

Ainsi, dans le document précédent, l'élément qui contient le nom de la machine est accessible par le chemin `/host/name`, et le type de son système d'exploitation par `/host/os/type`. Pour les attributs, il suffit de précéder leur nom d'une arobase : `/host/network/interface/@name`.

Avec un sous-chemin suffisamment non ambigu, un chemin flottant peut être utilisé à la place d'un chemin absolu, avec un double slash : `//os/version`.

Quand plusieurs éléments peuvent correspondre, il est possible de filtrer avec un sélecteur, noté entre crochets. Le sélecteur le plus simple est le numéro du nœud, comme dans un tableau, sauf que la numérotation commence à 1 : `//interface/address[1]/@type`. La sélection peut d'ailleurs aussi s'effectuer sur la valeur d'un attribut `//interface/address[@type="ipv6"]`, ou encore sur la partie texte de l'élément `//os/kernel[contains(., "Kernel")]`.

Ce dernier exemple montre que XPath propose une petite bibliothèque de fonctions permettant de réaliser des opérations simples sur les nœuds (`last()`, `position()`, `count()`, `id()`...), ainsi que des opérations numériques (`sum()`, `round()`...), booléennes (`not()`...) et sur les chaînes (`concat()`, `contains()`, `substring()`, `translate()`...).

Avec `XML::LibXML` et XPath, rechercher un nœud devient bien plus simple :

```
my ($addr_node) = $dom->findnodes(
   ➥'//interface/address[@type="ipv4"]');
```

`findnodes()` renvoyant des nœuds, l'API DOM reste parfaitement utilisable à ce niveau :

```
print $addr_node->textContent(), "\n";
```

La différence étant qu'il faut bien moins de code et qu'il est plus facile de relire une recherche réalisée en XPath qu'en pur DOM. Il est même possible d'accéder directement à la valeur :

```
my $ipv4 = $dom->findvalue('//interface/
   ➥address[@type="ipv4"]');
```

Quand la structure du document est bien connue et que le chemin est bien construit, cela permet de réduire le code au strict minimum.

Utiliser SAX

`XML::LibXML` supporte SAX1 et SAX2, dans les portages définis par `XML::SAX`, qui définit le standard en la matière dans le monde Perl. La documentation reconnaît que ce support n'est pas aussi avancé et complet que pour DOM. C'est peut-être en partie dû au fait que, SAX étant né dans le monde Java, il est très orienté objet et nécessite de créer une classe respectant une interface particulière pour réaliser le traitement du document. Si cette manière de faire est assez coutumière avec un langage comme Java, elle l'est beaucoup moins en Perl, où l'habitude est plutôt de spécifier les *callbacks* par référence de fonction.

La création d'une classe de traitement SAX reste tout de même une tâche assez simple, puisqu'il suffit d'hériter de `XML::SAX::Base` et de ne définir que les méthodes qui seront utiles au traitement :

```
package MySAXHandler;
use strict;
use parent "XML::SAX::Base";

sub start_element {
    my ($self, $elt) = @_;
    print "start_element: $elt->{Name}\n";
}

sub end_element {
    my ($self, $elt) = @_;
    print "end_element: $elt->{Name}\n";
}
```

Les noms de ces méthodes sont fixés par le standard SAX. Ici, `start_element` et `end_element` correspondent de manière assez évidente aux éléments ouvrants et fermants. L'exécution de ce bout de code sur le fichier XML d'exemple affiche :

```
start_element: host
start_element: name
end_element: name
start_element: network
start_element: interface
start_element: address
end_element: address
...
```

Pour rechercher un élément particulier, il suffit donc de filtrer sur son nom, éventuellement en faisant attention au contexte pour éviter les problèmes de collision. Le code est trivial et sans grand intérêt, car là où SAX prend son ampleur, c'est dans la notion assez générique de filtre qu'il autorise.

En effet, si un filtre SAX traite son flux XML et renvoie le résultat toujours sous forme XML, cette sortie peut être fournie en entrée à un nouveau filtre SAX. Il devient alors possible de chaîner ainsi plusieurs filtres les uns à la suite des autres, exactement comme c'est le cas avec les filtres de texte (`grep`, `sed`, `cut`, `awk`, etc) dans un *shell* Unix.

Le module `XML::SAX::Machines` permet ainsi de réaliser des chaînes (des *tubes*, dans la terminologie usuelle) de manière très naturelle. Bien que les filtres montrés dans cet exemple n'existent pas tous, cela permet d'illustrer la manière de construire un tel chaînage :

```
use XML::Filter::Grep;
use XML::Filter::Sort;
use XML::Filter::AsText;
use XML::SAX::Machines  qw< Pipeline >;

# ne garder que les titres de Kokia
# et Aural Vampire
my $grep = XML::Filter::Grep->new(
    '//author' => qr/Kokia¦Aural Vampire/,
);

# trier les titres par nom d'album
# et numéro de chanson
my $sort = XML::Filter::Sort->new(
     Record  => "track",
     Keys    => [
         [ "album", "alpha", "asc" ],
         [ "track-number", "num", "asc" ],
     ],
);

my $pipe = Pipeline(
     $grep => $sort => XML::Filter::AsText
      ➥   => \*STDOUT
);

$pipe->parse_file("music-library.xml");
```

Cet exemple permettrait, partant d'une hypothétique base de musiques, de filtrer pour ne garder que les titres des artistes Kokia et Aural Vampire, puis trierait les titres restants par nom d'album (un tri alphabétique) et numéro de chanson dans l'album (un tri numérique). Le résultat serait alors transformé en texte simple, plus lisible, qui serait alors affiché sur la sortie standard.

Cela permet d'illustrer comment fonctionne `XML::SAX::Pipeline`, qui accepte aussi bien un objet correspondant à un filtre SAX que le simple nom du module correspond (auquel cas il le charge automatiquement). Le premier et le dernier argument peuvent être des chemins à passer à la fonction `open()` de Perl, ou des descripteurs de fichiers. Le tout est assez magique et donc un peu perturbant à maîtriser, mais permet de réaliser des choses étonnantes.

`XML::SAX::Machines` propose aussi d'autres mécanismes de multitraitement, comme `XML::SAX::Manifold`, qui permet de répartir le traitement de différentes parties du document entre différents filtres, et de fusionner les résultats à la fin.

XML::Twig

Les deux approches utilisées pour manipuler du contenu XML, le DOM et le SAX, ont chacunes leurs avantages et inconvénients. `XML::Twig` est une solution qui mélange ces deux approches. Pour simplifier, l'utilisation typique de `XML::Twig` se fait comme suit : le contenu XML est parcouru séquentiellement grâce au *SAX parsing*, avec une liste de règles qui permettent d'identifier des éléments XML sur lesquels s'arrêter. Dès qu'un tel élément est trouvé, le parcours séquentiel est arrêté, et la branche qui part du nœud est chargée entière-

ment en mémoire. Il est alors possible de faire du *DOM parsing* sur cette partie du contenu.

Le module `XML::Twig` s'utilise en créant un objet instance, qui permet de charger du contenu, et d'effectuer toutes les opérations sur le document dans son ensemble.

Pour changer, supprimer ou ajouter du contenu, `XML::Twig` permet d'accéder à des objets éléments (de type `XML::Twig::Elt`). Ces objets éléments représentent un *tag* XML, et proposent une grande quantité de méthodes pour altérer le contenu du document.

Créer un objet XML::Twig

```
XML::Twig->new(. . .)
```

La méthode `new()` permet de créer un objet `XML::Twig`, indispensable à l'utilisation de `XML::Twig`.

```
my $t = XML::Twig->new()
```

Une option importante de cette méthode est `load_DTD`, qui permet de charger les informations de DTD.

```
my $t = XML::Twig->new(load_DTD => 1);
```

Lorsque cette option est définie à une valeur vraie, `XML::Twig` interprète les instructions relatives aux DTD et charge les fichiers DTD correspondants.

Charger du contenu XML avec XML::Twig

```
$t->parse()
$t->parsefile()
$t->parsefile_inplace()
```

La méthode `parse()` prend en argument soit une chaîne de caractères contenant le contenu XML à traiter, soit un objet de type `IO::Handle`, déjà ouvert (sur un fichier par exemple).

Dans la plupart des cas cependant, il est plus utile d'utiliser `parsefile()`, qui permet de lire le contenu depuis un fichier.

```
use XML::Twig;
my $t = XML::Twig->new();
$t->parse('<doc><elt couleur="rouge"/>
   ➥</doc>');

$t->parsefile('/un/fichier.xml');

$t->parsefile_inplace('/un/fichier.xml');
```

`parsefile_inplace` charge un document XML depuis un fichier ; cependant ce même fichier servira également à stocker le contenu final modifié. Cette méthode est utile pour faire des modifications sur un fichier XML.

Info

Cet exemple de code ne produit rien, et c'est normal : contrairement à `XML::LibXML`, les méthodes `parse` et `parsefile` ne renvoient rien. Ici, il est juste demandé au module `XML::Twig` de charger et parcourir le contenu XML, mais aucune action ou règle n'a été donnée, donc aucun résultat n'est produit.

Il faut en effet créer au moins un gestionnaire (*handler*, voir la section suivante) pour interagir avec le document XML.

Créer des handlers avec XML::Twig

$t->setTwigHandlers(..)

Pour pouvoir interagir avec le document XML, il faut créer des gestionnaires (*handlers*). Ce sont des règles qui associent une expression de type XPath à une référence sur une fonction. Voici un exemple de gestionnaire :

```
{
'liste/pays'        => \&fonction1 ,
'ville[@capitale]' => \&fonction2 ,
}
```

Le principe est simple : lors du parcours du document XML, chaque élément (tag XML) parcouru est comparé aux expressions XPath des gestionnaires. Lorsqu'une règle correspond, la fonction associée est évaluée. Celle-ci reçoit en arguments :

- Argument 1 : l'objet `XML::Twig`, (ici `$t`).
- Argument 2 : un nouvel objet `XML::Twig::Elt` (ici `$elt`), qui correspond à l'élément qui a vérifié la règle du gestionnaire.

Dans l'exemple précédent, si un tag XML a pour nom `pays` et comme parent `liste`, alors il vérifie la première règle XPath et la méthode `f1()` est appelée. De manière semblable, si un élément XML a pour nom `ville` et comporte un attribut `capitale`, alors `f2()` est appelée.

Voici un exemple fonctionnel avec ces gestionnaires :

```
use XML::Twig;
my $t = XML::Twig->new();
my $xml = '<doc><liste>' .
  '<pays nom="France">' .
    '<ville nom="Paris" capitale="1"/>' .
    '<ville nom="Lyon"/>' .
  '</pays>' .
  '</liste></doc>';
my $handlers = {
  'liste/pays'        => \&f1,
  'ville[@capitale]' => \&f2,
};
$t->setTwigHandlers($handlers);
$t->parse($xml);

sub f1 { my ($t, $elt) = @_; say "pays"}
sub f2 { my ($t, $elt) = @_; say "capitale"}
```

Attention

Il est important d'appeler `setTwigHandlers` *avant* `parse` ou `parsefile`. En effet, les gestionnaires doivent être en place avant de parcourir le document XML.

Cet extrait de code affiche :

```
capitale
pays
```

En effet, il y a un élément `pays` qui correspond à la première règle, et un élément `ville` (Paris) qui correspond à la seconde.

Info

La règle XPath `ville[@capitale]` est déclenchée *avant* la règle `liste/pays` car `XML::Twig` attend le tag de fin d'un élément avant d'examiner les règles. Or, l'élément `ville` est à l'intérieur de l'élément `pays`, donc il se ferme avant. `XML::Twig` déclenche les évaluations des règles les plus profondes d'abord : il effectue un *parcours en profondeur* dans l'arbre XML.

Produire le contenu XML de sortie

```
$t->flush()
$elt->flush()
```

Pour pouvoir produire le contenu XML du document, la méthode `flush` peut être utilisée sur l'objet `XML::Twig` mais également sur l'objet élément `XML::Twig::Elt` (reçu en argument des fonctions du gestionnaire).

`flush` va produire le contenu XML des éléments qui ont été parcourus, et décharger la mémoire. Il est donc possible de parcourir de très gros fichiers en *flush*-ant régulièrement le contenu.

L'exemple suivant utilise `flush` sur l'objet `XML::Twig` (voir page 291 pour des exemples qui utilisent `flush` sur des objets éléments de type `XML::Twig::Elt`).

```
use XML::Twig;
my $t = XML::Twig->new();
$t->parse('<doc><elt couleur="rouge"/>
   ➥</doc>');
$t->flush(); # affiche le XML en sortie
```

Ignorer le contenu XML de sortie

```
$t->purge()
$elt->purge()
```

Il arrive souvent que la tâche à accomplir sur un document XML soit en *lecture seule*, c'est-à-dire que les opéra-

tions vont consister uniquement à récupérer des informations du document, pas à modifier le document lui-même. Ainsi, le contenu XML du document n'est pas intéressant. Pour signifier à `XML::Twig` que le contenu examiné jusqu'ici peut être déchargé de la mémoire, il suffit d'utiliser `purge`.

Accéder au nom d'un élément

```
$elt->name()
```

Une fois qu'un gestionnaire a été mis en place, il est important de pouvoir accéder aux données de l'élément qui a vérifié la règle.

`name()` permet d'obtenir le nom du tag XML qui a vérifié la règle. Voici un exemple qui illustre l'utilisation de `name()`.

```
use XML::Twig;
my $t = XML::Twig->new();
$t->setTwigHandlers({ 'villes/*' => \&f1 });
$t->parse('<villes><Paris/><Lyon/></villes>'
    );

sub f1 { my($t, $elt) = @_; # récupère
                            # les paramètres
         say $elt->name(); # affiche le nom
       }
```

Ce programme affiche :

```
Paris
Lyon
```

Attention

La méthode `name` et les suivantes s'appliquent sur un objet `XML::Twig::Elt`, obtenu en général en argument de la fonction associée au gestionnaire. Une erreur très fréquente est d'essayer d'utiliser ces méthodes sur l'objet `XML::Twig`, ce qui ne fonctionnera pas

Changer le nom d'un élément

```
$elt->set_name(..)
```

Pour changer le nom d'un tag XML, il suffit d'utiliser `set_name()` :

```
use XML::Twig;
my $t = XML::Twig->new();
$t->setTwigHandlers({ 'villes/*' => \&f1 });
$t->parse('<villes><Paris/><Lyon/></villes>'
    );

sub f1 { my ($t, $elt) = @_; # récupère les
                             # paramètres
    $elt->set_name("Nancy"); # change le nom
    $elt->flush();
  }
$t->flush();
```

Ce qui donne en sortie :

```
<villes><Nancy/><Nancy/></villes>
```

Obtenir le contenu texte d'un élément

```
$elt->text()
$elt->text_only()
$elt->trimmed_text();
```

La méthode `text()` renvoie le contenu PCDATA et CDATA de l'élément. Les tags XML ne sont pas retournés, seul leur contenu texte l'est. Cette méthode renvoie également le contenu des éléments enfants.

`text_only` se comporte de la même manière, mais seul le contenu de l'élément est renvoyé, pas ceux de ses enfants.

`trimmed_text` est identique à `text()`, à la différence que les espaces superflues sont retirées de la chaîne de caractères retournée.

Astuce

La chaîne `'no_recurse'` peut être passée en argument pour se restreindre au contenu de l'élément, et ne pas inclure ceux des enfants de l'élément.

Changer le contenu XML d'un élément

```
$elt->set_inner_xml(..)
```

Dans certains cas, il est utile de remplacer complètement le contenu d'un élément. Il est possible de donner une chaîne XML à l'objet `XML::Twig::Elt` pour remplacer son

contenu (sans changer le tag de l'élément lui-même), grâce à la méthode `set_inner_xml` :

```
use XML::Twig;
my $t = XML::Twig->new();
$t->setTwigHandlers({ 'villes' => \&f1 });
$t->parse('<villes></villes>');

sub f1 {
  my ($t, $elt) = @_; # récupère les param.
  $elt->set_inner_xml(
    '<ville name="Paris"/>'
  );
}
$t->flush();
```

Le résultat est le suivant :

```
<villes><ville name="Paris"/></villes>
```

Interagir avec les attributs d'un élément

```
$elt->att()
$elt->set_att(..)
$elt->del_att() . . .
```

Il est très facile de manipuler les attributs d'un élément `XML::Twig::Elt`, en utilisant les méthodes suivantes :

- `att($nom_attribut)` : cette méthode renvoie la valeur de l'attribut dont le nom est `$nom_attribut`.
- `set_att($nom_attribut, $valeur_attribut)` : cette méthode change la valeur de l'attribut dont le nom est `$nom_attribut`, à la valeur `$valeur_attribut`.

- `del_att($nom_attribut)` : cette méthode supprime l'attribut dont le nom est `$nom_attribut`.
- `att_exists($nom_attribut)` : cette méthode renvoie vrai si l'élément contient un attribut dont le nom est `$nom_attribut`.

Voir page 295 pour un exemple de code illustrant l'utilisation de ces méthodes.

Interagir avec les éléments environnants

```
$elt->root()
$elt->parent()
$elt->children(), . . .
```

Lorsqu'une règle du gestionnaire est vérifiée, `XML::Twig` charge l'élément qui vérifie la règle et donne accès à l'arbre de tous les éléments qui n'ont pas encore été *flush*-és ou *purge*-és.

Ainsi, `XML::Twig` offre un véritable support DOM, et fournit une liste conséquente de méthodes pour naviguer dans l'arbre du document, en partant de l'élément, ou de la racine. En voici une sélection :

- `$elt-root()` : cette méthode renvoie un objet `XML::Twig::Elt` qui correspond à la racine du document XML.
- `$elt-parent()` : cette méthode renvoie un objet `XML::Twig::Elt` qui correspond au parent de l'élément en cours.
- `$elt-children()` : cette méthode renvoie la liste des enfants de l'élément. Les éléments enfants sont également de type `XML::Twig::Elt`.

- `$elt-ancestors()` : cette méthode renvoie la liste des « ancêtres » de l'élément. Cette liste contient le parent de l'élément, le parent du parent, etc., jusqu'à la racine du document.
- `prev_sibling` et `prev_siblings` : `prev_sibling` renvoie l'élément précédant l'élément en cours, au même niveau de l'arbre XML[2]. `prev_siblings` renvoie la liste des éléments de même niveau précédant l'élément en cours.
- `next_sibling` et `next_siblings` : ces méthodes sont les pendants de `prev_sibling` et `prev_siblings`, et renvoient le « frère » suivant, ou la liste des « frères » suivants.

Info

Toutes ces méthodes peuvent prendre en paramètre une expression XPath optionnelle, qui permet de filtrer les résultats. Ainsi, pour obtenir tous les enfants d'un élément donné qui ont l'attribut `couleur` à la valeur `bleu`, il suffit d'écrire :

```
$elt->children('*[@couleur = "bleu"]');
```

Cet extrait de code utilise les méthodes précédentes pour manipuler un document XML complexe :

```
use XML::Twig;
my $t = XML::Twig->new();
my $xml = '<doc><liste>' .
  '<pays nom="France">' .
    '<ville nom="Paris" capitale="1"/>' .
    '<ville nom="Lyon"/>' .
  '</pays>' .
  '<pays nom="Allemagne">' .
    '<ville nom="Berlin" capitale="1">' .
      'Ceci est Berlin' .
```

2. *Sibling* veut dire « frères et sœurs ».

```
    '</ville>' .
    '<ville nom="Francfort" capitale="0"/>' .
  '</pays>' .
  '</liste></doc>';
my $handlers = {
  'pays'        => \&f1,
};
$t->setTwigHandlers($handlers);
$t->parse($xml);

sub f1 {
  my ($t, $elt) = @_;
  say "racine : " . $elt->root->name;
  say $elt->name . ': ' . $elt->att('nom');
  my @enfants = $elt->children;
  say "  nb enfants : " . scalar(@enfants);
  foreach (@enfants) {
    say "  - " . $_->name . ': ' .
    ➥ $_->att('nom');
    say "    est capitale" if $_->att(
    ➥ 'capitale');
    my $texte = $_->trimmed_text();
    say "    '$texte'" if length $texte;
  }
  $elt->purge;
}
```

Ce programme affiche :

```
racine : doc
pays: France
  nb enfants : 2
  - ville: Paris
    est capitale
  - ville: Lyon
racine : doc
pays: Allemagne
  nb enfants : 2
  - ville: Berlin
    est capitale
    'Ceci est Berlin'
  - ville: Francfort
```

Effectuer un copier-coller

```
$elt->copy() $elt->paste(..)
```

La puissance de `XML::Twig` s'illustre dans une fonctionnalité majeure de ce module : le *copier-coller*.

La méthode `copy` renvoie une copie profonde de l'élément, c'est-à-dire que tous les éléments enfants sont également copiés.

La méthode `cut` coupe l'élément. Dorénavant, il n'est plus attaché à l'arbre DOM, il n'appartient plus au document XML. Cependant il peut être utilisé, notamment en argument à la méthode `paste`.

La méthode `paste` permet de coller un élément qui a été précédemment *copié* ou *coupé*. `paste` prend en argument la position à laquelle coller, et l'élément à coller. La position peut être :

- `before` : l'élément à coller est alors placé avant l'élément courant.
- `after` : l'élément à coller est alors placé après l'élément courant.
- `first_child` : l'élément à coller est inséré comme premier enfant de l'élément courant.
- `last_child` : l'élément à coller est inséré comme dernier enfant de l'élément courant.

Attention

La méthode `paste()` doit être appliquée sur l'élément source, qui a été copié ou coupé, et non sur l'élément cible. L'élément cible doit être passé en argument.

Voici un exemple :

```
use XML::Twig;
my $t = XML::Twig->new();
my $xml = '<doc><liste>' .
 '<pays nom="France">' .
   '<ville nom="Paris" capitale="1"/>' .
   '<ville nom="Lyon" capitale="0"/>' .
 '</pays>' .
 '<pays nom="Allemagne">' .
   '<ville nom="Francfort" capitale="0"/>' .
 '</pays>' .
 '</liste></doc>';
my $handlers = {
 'pays[@nom="Allemagne"]' => \&f1,
};
$t->setTwigHandlers($handlers);
$t->parse($xml);
$t->flush();

sub f1 {
 my ($t, $allemagne) = @_;
 my $francfort=($allemagne ->children())[0];
 say $francfort->att('nom');
 $francfort->cut();
 my $france = $allemagne->prev_sibling();
 $francfort->paste(last_child => $france);
}
```

Ce programme produit le document XML suivant :

```
<doc>
  <liste>
    <pays nom="France">
      <ville nom="Paris" capitale="1" />
      <ville nom="Lyon" capitale="0" />
      <ville nom="Francfort" capitale="0"/>
    </pays>
    <pays nom="Allemagne">
    </pays>
  </liste>
</doc>
```

Autres références

`XML::Twig` implémente énormément de méthodes, certaines étant des synonymes d'autres (ainsi `$elt-name()` est équivalent à `$elt-tag()` et `$elt-gi()`). Cet aspect apporte beaucoup de puissance, mais est quelque peu déroutant pour le débutant. C'est pourquoi ce présent chapitre s'est concentré sur une sélection des concepts et fonctions majeurs de `XML::Twig`.

Pour acquérir une meilleure maîtrise du module `XML::Twig`, consultez sa documentation exhaustive sur CPAN : **http://search.cpan.org/perldoc?XML$::$Twig.**

15

Sérialisation de données

La *sérialisation* est l'opération consistant à transformer une structure de données en mémoire en un format pouvant être transféré ou enregistré sur disque, afin de pouvoir relire ce format (on dit *désérialiser*) pour reconstituer la structure plus tard, dans le même programme, ou dans un autre programme. Cela permet par exemple d'échanger des données complexes entre des programmes, ou de sauvegarder l'état mémoire pour une reprise rapide sur panne. De nombreux formats existent et correspondent à des types de besoins différents.

Nous examinerons en premier lieu les formats natifs, puis JSON et YAML.

Sérialiser avec `Data::Dumper`

```
use Data::Dumper;
```

Perl intègre déjà en standard des modules permettant de sérialiser des structures. Le plus simple est `Data::Dumper` qui génère le code Perl permettant de reconstruire la structure par une simple évaluation.

```
use Data::Dumper;

my %contact = (
  dams => {
    name  => "Damien Krotkine",
    email => "...",
  },
  jq => {
    name  => "Jérôme Quelin",
    email => "...",
  },
  book => {
    name  => "Philippe Bruhat",
    email => "...",
  },
  maddingue => {
    name  => "Sébastien Aperghis-Tramoni",
    email => "...",
  },
);

my $serialised = Dumper(\%contact);
print $serialised;
```

Cet exemple affiche :

```
$VAR1 = {
  'jq' => {
    'email' => '...',
    'name' => 'Jérôme Quelin'
  },
  'dams' => {
    'email' => '...',
    'name' => 'Damien Krotkine'
  },
  'maddingue' => {
    'email' => '...',
    'name' => 'Sébastien Aperghis-Tramoni'
    },
  'book' => {
```

```
    'email' => '...',
    'name' => 'Philippe Bruhat'
  }
};
```

Pour relire la structure en mémoire, il suffit d'évaluer la chaîne ou le fichier dans lequel le résultat de `Data::Dumper` a été stocké, avec respectivement `eval` ou `do`.

```
eval $serialised;
```

`Data::Dumper` est suffisamment intelligent pour détecter si une sous-structure est utilisée plus d'une fois, et il emploiera des références pour éviter de dupliquer les données.

```
my %contact = (
  dams => {
    name  => "Damien Krotkine",
    email => "...",
  },
  jq   => {
    name  => "Jérôme Quelin",
    email => "...",
  },
  book => {
    name  => "Philippe Bruhat",
    email => "...",
  },
  maddingue => {
    name  => "Sébastien Aperghis-Tramoni",
    email => "...",
  },
);

my %project_info = (
  perlbook1 => {
    name    => "Perl Moderne",
    manager => $contact{dams},
```

```
    team     => [
      @contact{qw< dams jq book maddingue >}
    ],
  },
  Curses_Toolkit => {
    name     => "Curses::Toolkit",
    manager => $contact{dams},
    team     => [
      @contact{qw< dams jq maddingue >}
    ],
  },
);
```

La sérialisation de `%project_info` donne le résultat suivant :

```
$VAR1 = {
  'perlbook1' => {
    'name' => 'xxx',
    'team' => [
      {
        'email' => '...',
        'name' => 'Damien Krotkine'
      },
      {
        'email' => '...',
        'name' => 'Jérôme Quelin'
      },
      {
        'email' => '...',
        'name' => 'Philippe Bruhat'
      },
      {
        'email' => '...',
        'name' => 'Sébastien Aperghis-Tramoni'
      }
    ],
    'manager' => $VAR1->{perlbook1}{team}[0]
  },
```

```
  'Curses_Toolkit' => {
    'name' => 'Curses::Toolkit',
    'team' => [
      $VAR1->{'perlbook1'}{'team'}[0],
      $VAR1->{'perlbook1'}{'team'}[1],
      $VAR1->{'perlbook1'}{'team'}[3]
    ],
    'manager' => $VAR1->{perlbook1}{team}[0]
  }
};
```

La structure est correctement sauvegardée de façon à être autonome, tout en préservant ses liens internes.

Si les deux structures sont sérialisées en même temps, `Data::Dumper` voit mieux les liens, conservant une meilleure cohérence globale :

```
print Dumper(\%contact, \%project_info);
```

affiche :

```
$VAR1 = {
  # contenu de %contact
};
$VAR2 = {
  'perlbook1' => {
    'name' => 'xxx',
    'team' => [
      $VAR1->{'dams'},
      $VAR1->{'jq'},
      $VAR1->{'book'},
      $VAR1->{'maddingue'}
    ],
    'manager' => $VAR1->{'dams'}
  },
  # etc.
};
```

Mais ces exemples mettent en évidence certains aspects ennuyeux de `Data::Dumper`. En premier lieu, le fait qu'il affecte les données à des variables aux noms courts mais disgracieux `$VAR1`, `$VAR2`, etc. En mode strict, il faut que ces noms existent, ce qui est plus que gênant. Cela peut se contourner en utilisant une forme plus élaborée d'appel de `Data::Dumper`, qui permet de donner des noms plus explicites :

```
print Data::Dumper->Dump(
  [ \%contact, \%project_info ],
  [qw< contact project_info >]
);
```

affiche :

```
$contact = {
  # contenu de %contact
};
$project_info = {
  # contenu de %project_info
};
```

C'est mieux, mais cela impose tout de même que ces variables devront exister dans le contexte du programme qui chargera ces données.

Un autre problème assez évident est que, du côté du programme qui va charger les données, il va véritablement falloir évaluer du code Perl étranger, ce qui est évidemment dangereux. Hormis dans les cas où les modules présentés ci-après ne seraient pas utilisables, `Data::Dumper` est donc à déconseiller comme solution de sérialisation, mais s'avère un précieux partenaire de déboggage.

Sérialiser avec `Storable`

```
use Storable;
```

Perl propose un autre module pour la sérialisation, `Storable`. La différence majeure avec `Data::Dumper` est que `Storable`, écrit en XS, sérialise sous forme binaire, ce qui offre un stockage beaucoup plus compact et évite les problèmes liés à l'évaluation du code généré par `Data::Dumper`. Son API, extrêmement simple, rend son utilisation plus agréable :

```
# copie profonde de %contact
my $serialised = freeze(\%contact);
my $contacts   = thaw($serialised);

# sauve dans le fichier
store(\%project_info, "projects.dmp");

# restaure depuis le fichier
my $projects = retrieve("projects.dmp");
```

Ces fonctions existent aussi en versions portables, qui stockent les octets dans l'ordre dit « réseau » : `nfreeze()`, `nstore()`.

Toutefois, qui dit format binaire, dit problème de compatibilité. `Storable` ne fait pas exception mais chaque version sait bien sûr lire les formats générés par les versions précédentes, et les versions récentes savent se montrer plus conciliantes. Globalement, `Storable` fonctionne comme on s'y attend.

Rapide, bien plus sûr, `Storable` constitue donc une solution de sérialisation à recommander pour répondre aux problèmes de stockage des données en local ou de transfert de structures et objets entre programmes Perl. Bien

évidemment, dès qu'il faut communiquer avec des programmes écrits dans d'autres langages, `Storable`, spécifique à Perl, devient inapproprié, et il faut recourir à JSON ou YAML.

JSON

JSON (*JavaScript Object Notation*) est un format qui dérive en fait de la syntaxe normale de JavaScript pour construire les structures, limitée à la seule sous-syntaxe pour réaliser cette tâche spécifique, pour d'évidentes raisons de sécurité. La syntaxe est très naturelle pour un programmeur Perl :

- Les tableaux sont déclarés par des crochets, avec les valeurs séparées par des virgules : `[ "value", . . .]`.
- Les tables de hachages (la norme JSON parle d'objets) sont déclarées par des accolades, avec les clés séparées de leur valeur par un deux-points et les couples clé-valeur séparés entre eux par des virgules : `{ key: "value", . . .}`.
- Les booléens sont déclarés avec les littéraux `true` et `false`.
- La valeur `undef` se traduit par le littéral `null`.
- Les nombres sont reconnus sous forme décimale, avec partie fractionnelle et notation exponentielle :

```
42          # valide
3.1415269   # valide
-273.15     # valide
3.086e16    # valide

0755        # NON valide (octal)
0xdeadbeef  # NON valide (hexadécimal)
```

- Toute autre valeur doit être sous forme de chaîne, entre guillemets, avec les règles usuelles des chaînes en C :

```
"The quick brown fox jumps over the lazy dog.\n"
"c:\\windows\\system"
```

JSON est par défaut en UTF-8. Les caractères Unicode du plan de base multilingue (entre U+0000 et U+FFFF) peuvent s'écrire sous la forme \uXXXX où XXXX est la valeur hexadécimale de quatre chiffres correspondant au code-point du caractère.

```
"J\u00e9r\u00f4me va \u00e0 la p\u00eache
en sifflant \u266A"
```

Si le caractère n'est pas dans le plan de base, il suffit d'utiliser les paires de seizets d'indirection (*surrogate pairs*).

JSON est donc un format assez simple, léger, qui permet de représenter de manière directe, et sans trop de réflexion, les structures typiques rencontrées aussi bien en Perl que dans la plupart des autres langages dynamiques. Il est donc très utile pour échanger des données complexes entre des programmes écrits dans des langages différents.

Plus encore, il est parfaitement adapté au contexte du développement web, pour servir d'intermédiaire entre les programmes serveurs en Perl et l'interface en HTML et JavaScript, où l'évaluation du JSON est bien plus rapide que l'analyse du XML, et son exploitation bien plus aisée. Raison pour laquelle il a remplacé depuis longtemps ce dernier comme format de prédilection dans les techniques dites Ajax.

Sérialiser avec JSON

```
use JSON;
```

Bien évidemment, il existe en Perl des modules appropriés pour générer et lire le JSON sans problème. Les deux plus importants à retenir sont `JSON` (dénommé `JSON.pm` dans la suite pour mieux le distinguer du format lui-même) et `JSON::XS`. La différence entre les deux est que le premier est écrit en pur Perl alors que le second est en C, et donc bien plus rapide. Mais `JSON.pm` sait utiliser `JSON::XS` s'il est présent, ce qui simplifie la tâche.

Dans sa forme la plus simple, ce module s'utilise ainsi :

```
use JSON;

my $json_text   = to_json(\%struct);
my $perl_struct = from_json($json_text);
```

Ainsi, avec notre hash de contacts :

```
my $json_text   = to_json(\%contact);
print $json;
```

cela affiche :

```
{"jq":{"email":"...",
    "name":"Jérôme Quelin"},
"dams":{"email":"...",
     "name":"Damien Krotkine"},
"maddingue":{"email":"...",
     "name":"Sébastien Aperghis-Tramoni"},
"book":{"email":"...",
     "name":"Philippe Bruhat"}}
```

Les sauts de ligne ont ici été ajoutés pour rendre la sortie plus lisible, mais JSON se voulant un format très compact,

il est par défaut généré sans espace ni saut de ligne. Cela peut se changer en passant des options à ces fonctions, sous forme de référence à un hash :

```
my $json_text   = to_json(\%struct,
   ➥{ ... } );
my $perl_struct = from_json($json_text,
   ➥{ ... });
```

Les options les plus intéressantes sont :

- `ascii` : tous les caractères Unicode hors ASCII seront transcrits sous la forme `\uXXXX`, afin que le résultat soit parfaitement compatible avec ASCII (7 bits).
- `latin1` : l'encodage se fera en ISO-Latin-1 (ISO-8859-1), les caractères en dehors de ISO-Latin-1 seront transcris sous la forme `\uXXXX`.
- `utf8` : l'encodage se fera en UTF-8.
- `pretty` : équivalente aux options `indent`, `space_before` et `space_after`, permet un affichage plus lisible pour les humains.
- `relaxed` : autorise le décodage de textes qui ne sont pas totalement conformes à la norme JSON, dans certaines limites, ce qui peut être utile pour relire des données écrites par des humains ou par un mauvais générateur.

En reprenant l'exemple précédent et en ajoutant quelques options :

```
my $json_text = to_json(
  \%contact, { pretty => 1, ascii => 1 }
);
print $json_text;
```

affiche :

```
{
  "jq" : {
    "email" : "...",
    "name" : "J\u00e9r\u00f4me Quelin"
  },
  ...
}
```

Comme il serait ennuyeux de devoir passer ces options à chaque appel de fonction, `JSON.pm` permet de créer des objets afin d'avoir son convertisseur JSON personnalisé.

```
my $json = JSON->new;

# les modificateurs d'options renvoient
# l'objet, ce qui permet de les chaîner
$json->pretty(1)->space_before(0);
$json->ascii(1);

my $json_text = $json->encode(\%contact);
print $json_text;
```

affiche :

```
{
  "jq": {
    "email": "...",
    "name": "Jérôme Quelin"
  },
  ...
}
```

Si créer un objet est moyennement intéressant pour les options assez simples présentées ici, cela devient par contre indispensable pour exploiter les options avancées telles que les filtres ou les mécanismes de gestion des données spéciales et inconnues.

À noter que les exemples donnés ici utilisent l'API de la version 2.xx du module `JSON.pm`, incompatible avec celle

de la précédente version. Même si c'est fondamentalement pour le mieux (fonctions mieux nommées, meilleure API objet, plus aucune variable globale), il est fâcheux que l'auteur n'ait pas intégré un moyen pour permettre une meilleure transition de la v1 à la v2. Pour des usages basiques, les quelques lignes suivantes permettent d'utiliser l'API de `JSON.pm` v2 y compris avec la v1 :

```
# use JSON v2 API even with JSON v1
if ( JSON->VERSION < 2.00 ) {
   no warnings;
   *to_json = *JSON::encode = *JSON::to_json
        = \&JSON::objToJson;
   *from_json = *JSON::decode = *JSON::
        from_json
        = \&JSON::jsonToObj;
}
```

Le cas typique d'un tel besoin, rencontré par l'un des auteurs, est que l'environnement de production ne peut souvent pas utiliser les versions très récentes des modules, de par leur délai d'intégration dans les distributions.

YAML

Par rapport à JSON, YAML n'est pas tant un *format* de sérialisation qu'un véritable *langage* de sérialisation. Conçu par des développeurs chevronnés des langages dynamiques Perl, Python et Ruby, il est disponible pour tous les langages majeurs, ce qui le rend très intéressant pour sérialiser des données complexes de manière portable.

Ses capacités viennent toutefois avec un coût, celui de la complexité de sa spécification. Elle ne sera donc examinée ici que dans ses aspects les plus simples. Ce n'est toutefois pas très gênant pour commencer car du point de vue de l'utilisateur,

cela reste transparent, et surtout YAML est bien plus lisible que JSON.

YAML possède deux modes de syntaxe : par bloc et en ligne, qui peuvent se mélanger sans grandes restrictions. La syntaxe en ligne est fondamentalement du JSON, ce qui permet à tout analyseur YAML de pouvoir relire sans problème un fichier JSON. La syntaxe en mode bloc sépare chaque donnée sur sa ligne propre, et repose fortement sur l'indentation pour déterminer en partie la profondeur des structures.

- Les commentaires commencent par un dièse #.
- Les valeurs sont reconnues de façon assez naturelle par l'analyseur YAML, et hormis quand elles contiennent des caractères faisant partie de la syntaxe, il n'est même pas nécessaire d'encadrer les chaînes entre guillemets.

```
# pas besoin de guillemets
The quick brown fox jumps over the lazy dog

# par contre, dans ce cas, oui
"Le monsieur te demande : où es-tu ?"
```

- Les tableaux s'appellent des *séquences* et sont notés en mode ligne par des crochets, les valeurs étant séparées par des virgules :

```
[ Sébastien, 6, Antibes ]
```

En mode bloc, les données sont simplement introduites par un tiret sur la colonne courante :

```
- Damien
- Jérôme
- Philippe
- Sébastien
```

- Les tables de hachages s'appellent des *correspondances* (*mappings*) et sont notées en mode ligne par des accolades, les clés séparées de leur valeur par un deux-points et les couples clé-valeur séparés entre eux par des virgules :

```
{name: Damien, dept: 94, city: Vincennes}
```

  En mode bloc, les couples clé-valeur sont simplement chacun sur leur ligne propre :

```
name: Philippe
dept: 69
city: Villeurbanne
```

  YAML impose que chaque clé soit unique au sein d'une correspondance.

Bien évidemment, ces éléments peuvent s'imbriquer pour construire des structures complexes :

```
perlbook1:
- title: Perl Moderne
- publisher: Pearson
- {name: Damien, dept: 94, city: Vincennes}
- {name: Jérôme, dept: 69, city: Lyon}
- {name: Philippe, dept: 69, city: Villeurbanne}
- {name: Sébastien, dept: 6, city: Antibes}
```

Un point important est qu'un fichier YAML peut contenir plusieurs documents internes. Un document commence par trois tirets - - -, et continue jusqu'au document suivant, jusqu'à une fin de document, marquée par trois points . . . ou jusqu'à la fin du fichier.

Sérialiser avec YAML

```
use YAML;
```

Sans surprise, de nombreux modules sont disponibles sur le CPAN, mais il n'est véritablement nécessaire d'en connaître qu'un seul, tout simplement nommé `YAML`, qui correspond à l'implémentation standard, en pur Perl de l'analyseur YAML. Parmi les autres modules qui méritent d'être mentionnés se trouvent `YAML::Syck` qui utilise la bibliothèque C `libsyck` (supporte YAML 1.0), et `YAML::XS` qui utilise la `libyaml` (supporte YAML 1.1). Le bon point est que `YAML` détecte seul la présence d'un de ces modules, et le charge automatiquement afin de bénéficier de meilleures performances.

Dans tous les cas, l'interface d'utilisation reste la même, et ressemble un peu à celle de `Data::Dumper`. Ainsi, pour transformer des structures en documents YAML, il suffit d'utiliser la fonction `Dump()` :

```
use YAML;

my %book = (
  title => "Perl Moderne",
  publisher => "Pearson",
);

my %contact = (
  # toujours les mêmes informations..
);

print Dump(\%book, \%contact);
```

affiche :

```
---
publisher: Pearson
title: Perl Moderne
---
book:
  email: ...
  name: Philippe Bruhat
dams:
  email: ...
  name: Damien Krotkine
jq:
  email: ...
  name: Jérôme Quelin
maddingue:
  email: ...
  name: Sébastien Aperghis-Tramoni
```

qui contient deux documents YAML, puisque deux structures ont été fournies. La fonction `DumpFile()` permet de sauver le flux YAML directement dans un fichier :

```
use YAML qw< DumpFile >;

DumpFile("perlbook1.yaml", \%book, \%contact
   ➥);
```

Charger un flux YAML utilise les fonctions inverses `Load()` et `LoadFile()`, qui renvoient logiquement autant de références qu'il y a de documents dans le flux :

```
use YAML qw< LoadFile >;

my ($book, $contact) =
   ➥LoadFile("perlbook1.yaml");
```

Le module dispose bien sûr de son lot d'options diverses, mais il est rare de devoir y faire appel.

Il faut noter que YAML permet aussi de sérialiser des objets, mais il faut qu'ils disposent des méthodes adéquates.

16

Fichiers de configuration

Quoi de plus banal que de pouvoir configurer un programme ? Pourtant, c'est trop souvent un point laissé de côté durant le développement, ajouté au dernier moment, lorsque le temps manque, ce qui aboutit bien fréquemment à cette horreur :

```
do "$ENV{HOME}/etc/monprog.pl";
```

Un fichier de configuration est un fichier de données mortes, qui doivent donc impérativement rester séparées du code.

Deux grands cas d'utilisation se distinguent : d'un côté, la simple affectation de paramètres, pour lequel le format .INI est très bien adapté, de l'autre, la définition de structures complexes, pour lesquelles JSON (voir page 308) ou YAML (voir page 313) fournissent enfin une réponse universelle.

Fichiers .INI

```
use Config::IniFiles;
```

Le fichier .INI, popularisé par Microsoft, permet d'affecter des valeurs à des paramètres (`param = value`), eux-mêmes groupés au sein de sections dont le nom est donné entre crochet (`[section_name]`), et supporte des commentaires, commençant par un point-virgule (;) et jusqu'à la fin de la ligne.

Toutefois, n'ayant jamais été vraiment formalisé de manière normative, de nombreuses extensions ou variations de ce format existent au gré des différentes implémentations : support du dièse (#) comme début de commentaire, support de séquences d'échappement (\t, \n, \xNNNN...), support de continuation de ligne (avec une barre oblique inverse \ en fin de ligne), etc.

Sans surprise, de nombreux modules sont disponibles sur CPAN pour lire des fichiers .INI, mais `Config::IniFiles` est probablement le plus complet de tous, et supporte en particulier plusieurs extensions.

Son API n'est pas tout à fait orthodoxe, avec un mélange de méthodes nommées en minuscules et d'autres en *camel case*, mais elle est globalement agréable et pratique à utiliser, ce qui est tout de même le principal.

Avec ce fichier de configuration comme exemple :

```
[general]
stats_path = /var/cache/stats.yml

[network]
port    = 1984
address = localhost
timeout = 5
```

lire des paramètres se fait ainsi :

```
use Config::IniFiles;

my $config = Config::IniFiles->new(
  -file => "/etc/mysoft.conf"
);

my $port = $config->val(network => "port");
```

Point intéressant, la méthode `val()` permet de gérer directement des valeurs par défaut, ce qui évite de devoir le faire avec `//` ou son équivalent sur les vieux Perl :

```
my $port  =
  ➥$config->val(network => "port", 1984);
```

`Config::IniFile` a d'ailleurs une gestion intéressante des valeurs par défaut, permettant par exemple d'aller chercher celles-ci dans une section particulière, qui peut être indiquée avec l'option `-default` de `new()`. Curieusement, il considère par défaut les fichiers vides comme une erreur, à moins de passer l'option `-allowempty` à vrai.

Le module propose aussi d'hériter depuis un objet existant, spécifié avec l'option `-import` : les sections sont fusionnées, mais les paramètres provenant du fichier analysé ont priorité.

```
my $default = Config::IniFiles->new(
  -file => "/etc/mysoft.global.conf"
);

my $config  = Config::IniFiles->new(
  -import => $default,
  -file => "/etc/mysoft.local.conf",
);
```

D'autres options de `new()` permettent de contrôler les extensions de syntaxe précédemment mentionnées :

- `-allowcontinue` : active le support des continuations de ligne par le caractère antislash `"\"`.
- `-commentchar` et `-allowedcommentchars` : permettent de spécifier les caractères de début de commentaire, par défaut `"#"` et `";"`.
- `-nocase` : permet de gérer les sections et paramètres de manière insensible à la casse (c'est-à-dire sans distinction entre majuscules et minuscules).

Cette présentation rapide couvre déjà une bonne partie des usages courants, par exemple récupérer des valeurs depuis le fichier de configuration, mais `Config::IniFiles` offre une gestion complète des paramètres, sections et groupes (encore une extension au format), permettant de les ajouter ou les détruire à la volée. Et pour être pleinement utile, il permet aussi de sauver la nouvelle configuration sur disque.

Il propose enfin une interface `tie()` pour ceux qui préfèrent que tout ressemble à un hash :

```
tie my %config,
    "Config::IniFiles",
    -file => "/etc/mysoft.conf";

print $config{network}{port};
# affiche "1984"
```

Pour ne rien gâcher, ce module existe depuis déjà plusieurs années, d'où une bonne disponibilité sous forme de paquet pour votre distribution favorite, et il est toujours bien maintenu.

Utilisation de JSON/YAML

Une question qui peut naturellement se poser est, après tout, pourquoi ne pas utiliser JSON ou YAML comme format de configuration ? Comme ces formats peuvent « tout faire », autant s'en servir aussi dans ce cas-là.

Certes, mais il faut pousser la réflexion plus loin. Le problème de JSON et YAML est que leur syntaxe, quoique relativement simple (comparée à du XML par exemple), reste néanmoins plus exigeante que celle d'un fichier .INI, et les modules d'analyse de ces derniers pardonnent ainsi beaucoup plus de petites erreurs. Par ailleurs, si JSON et YAML sont assez faciles à lire, ils restent moins naturels à écrire et demandent donc plus de réflexion. JSON en particulier demande de respecter de nombreux détails de syntaxe qui le rend vraiment peu intéressant comme format de fichier de configuration, ce qui fera donc préférer YAML pour sa plus grande clarté.

Pour des fichiers de configuration courants, la structure section / clé / valeur est largement suffisante, et YAML n'apporte aucun gain. Par contre, il devient bien plus intéressant dès qu'il s'agit de gérer des données plus complexes, qui ne rentrent pas dans le cadre simpliste du couple clé-valeur, l'exemple typique étant le stockage de données arborescentes.

17

Principes généraux de POE

La *programmation événementielle*, ou programmation asynchrone, peut facilement dérouter le développeur qui n'a pas encore été confronté à cette manière de concevoir et d'écrire des programmes, mais elle est au cœur de nombreux problèmes actuels. Parmi les premiers exemples qui viennent en tête, il y a les interfaces utilisateurs, qu'elles soient graphiques ou en mode console ou encore les serveurs réseau qui doivent généralement être en mesure de répondre à plus d'un client à la fois. Dans tous les cas, il s'agit d'être en mesure de traiter des événements qui peuvent survenir à tout moment (connexion d'un client, clic sur un widget, etc.), ce qui est impossible à réaliser avec un programme classique à flot d'exécution continu.

Il faut donc savoir reconnaître la nature asynchrone d'un futur logiciel avant de commencer à effectivement coder pour éviter de douloureuses déconvenues par la suite. L'un des auteurs peut ainsi témoigner des conséquences que peut engendrer un choix trop rapide. Brossons rapidement le contexte : il fallait développer un couple de *démons*, un serveur, un client, devant dialoguer par réseau, chaque client se connectant sur l'ensemble des serveurs déployés. Chaque *démon* doit donc gérer un certain

nombre de connexions actives, une demi-douzaine dans le cas des clients, une trentaine dans le cas des serveurs. Par peur de ne pas avoir le temps de maîtriser `POE` (très peu de temps de développement avait été accordé), cet auteur s'est tourné vers des modules plus simples à comprendre, `IO::Socket` et `IO::Select`. Et en effet, il est facile d'obtenir ainsi un prototype fonctionnel. Mais une fois en production, les problèmes et bogues arrivent bien rapidement, et même des mois plus tard, certains problèmes « incompréhensibles » dus à une mauvaise gestion des sockets. Tout cela parce que le choix de réinventer sa propre roue[1], surtout quand le temps manque, est très rarement une bonne idée. La leçon à retenir est que le temps passé à apprendre un environnement comme `POE` doit se voir comme un investissement qui ne pourra que s'avérer profitable à long terme.

POE

`POE`[2] est un environnement de programmation asynchrone pour Perl. Le principe est d'avoir plusieurs tâches qui travaillent ensemble, d'où le nom de multitâche coopératif.

Cela rappellera peut-être de mauvais souvenirs à ceux qui ont connu Windows 98 (et ses prédécesseurs) ou Mac-OS Classic, qui étaient des systèmes d'exploitation de ce type. Ainsi, si un programme était bloqué, cela figeait l'ensemble du système d'exploitation. D'autre part, le partage de ressources (processeur, mémoire et I/O) était inefficace et injuste.

1. Ou plutôt, comme dirait Jean Forget, réinventer sa propre brosse à dents ; *cf.* **http://journeesperl.fr/2007/talk/916** pour la vidéo de sa présentation.

2. *Perl Object Environment*, bien que l'acronyme ait donné lieu à de nombreuses autres interprétations sérieuses, telles que *Parallel Object Executor* ou plus facétieuses comme *Part Of an Elephant* ou *Potatoes Of Eternity*.

Pourquoi alors utiliser le même paradigme ? Parce que mis en œuvre à travers une application, le problème est complètement différent. En effet, une application est écrite par un seul développeur, ou une équipe plus ou moins réduite. Dans tous les cas, il est (relativement) facile de respecter l'environnement et quelques règles de bon fonctionnement.

Cela réduit les inconvénients du multitâche coopératif, en révélant toute la puissance d'un environnement multitâche.

`POE` est donc un environnement multitâche, mais utilise pour cela un seul processus et un seul *thread* ; ce qui a comme conséquence immédiate que les applications sont facilement portables, même sur les systèmes qui ne disposent pas de l'appel système `fork` ou sur les interpréteurs Perl qui ne sont pas multithreadés.

Deuxième conséquence : les applications sont beaucoup plus faciles à écrire. Finis les IPC et autres verrous pour s'assurer l'accès à une ressource : comme une seule tâche est en cours, toute action est considérée comme étant atomique.

`POE` est donc analogue à des bibliothèques de gestion de threads comme `Pth` en C, mais la comparaison s'arrête là car `POE` est de plus haut niveau et surtout entièrement orienté objet.

Événements

Une application `POE` est une suite d'*événements* (*events*) que les tâches s'envoient. La documentation parle encore souvent d'états (*states*), du fait de la construction originelle sous forme de machine à états, qui a par la suite évolué. Un événement signale simplement qu'il s'est passé quelque chose, par exemple :

- un fichier est prêt ;
- une alarme a expiré ;
- une nouvelle connexion est arrivée ;
- ou n'importe quel autre événement indiquant que l'application a changé d'état.

Les gestionnaires d'événements sont des *callbacks*, c'est-à-dire des fonctions enregistrées pour être appelées lorsque ces événements se produisent.

Sessions

Les tâches sont appelées des *sessions* en `POE`. Une session a des ressources privées, tels que descripteurs de fichiers, variables, etc. Elle dispose en particulier d'une *pile* (*heap*), dans laquelle elle peut stocker ce qu'elle veut. Elle peut aussi s'envoyer des événements privés, et même avoir des sessions filles. Pour faire l'analogie avec un système d'exploitation, une session est donc comme un processus.

Une session reste en vie tant qu'il le faut, c'est-à-dire tant qu'elle a des événements à traiter. Quand elle n'a plus rien à faire, la session s'arrête. Ceci peut paraître un peu nébuleux, mais est en fait très simple : une session peut par exemple référencer une connexion, avec des événements pour réagir à ce qui se passe sur cette connexion. Tant que la connexion est active, la session reste en vie, car celle-ci peut générer des événements à traiter. Quand la connexion est fermée et libérée, alors la session n'a plus de but (la connexion ne peut plus générer d'événements) et est donc terminée par `POE`.

Le corollaire est qu'une session, lors de sa création, doit obligatoirement créer et référencer quelque chose qui va la maintenir en vie. Sinon, elle est sera aussitôt passée au ramasse-miettes de `POE`.

Le noyau POE

Le noyau est le cœur de POE. Il est chargé d'acheminer les événements, d'appeler le code devant les gérer, d'orchestrer les sessions, mais aussi de vérifier des conditions en générant des événements si besoin. Le noyau est en effet le gestionnaire du temps (c'est lui qui va savoir quand lancer une alarme) et le responsable des objets bas-niveau tels que les descripteurs de fichiers et de connexions.

Le noyau est implémenté dans le module POE::Kernel, et fournit la méthode run() qui est en fait la boucle principale du programme, ne retournant pas tant qu'il reste des tâches :

```
# lance le programme une fois
# les sessions créées
POE::Kernel->run;
```

Un mot sur les performances

POE est un *framework* avancé, très orienté objet, ce qui implique une perte d'efficacité comparé au procédural pur. Il reste cependant performant et devrait suffire pour la majorité des besoins, d'autant plus qu'il apporte de nombreux avantages. Nous verrons par la suite qu'il existe des extensions permettant d'obtenir de meilleures performances.

En parlant chiffres : pour une application gérant jusqu'à 1 000 messages par seconde (ce qui correspond à la plupart des applications), POE convient sans hésiter. Au delà de 10 000 messages par seconde, POE ne suffit plus, ou en tout cas plus tout seul (voir au Chapitre 19 consacré à POE distribué). Entre ces deux valeurs, une évaluation reste nécessaire.

18

POE en pratique

Créer une session POE

```
POE::Session->create
```

Une session `POE` est créée avec le constructeur de session. Celui-ci accepte divers arguments, dont le plus important est `inline_states` qui définit les événements[1] que la session s'attend à recevoir :

```
use POE;

POE::Session->create(
    inline_states => {
        _start => sub {
            say "session démarrée";
        },
    },
);

POE::Kernel->run;
```

1. Au début, Rocco Caputo voulait créer avec `POE` un générateur de machines à états, d'où le nom de l'argument.

Une session va être créée et donc planifiée par le noyau `POE` lorsque celui-ci sera lancé.

L'événement `_start` va automatiquement être appelé lors du démarrage de la session, après la création de la session elle-même. Puis, comme la session n'a pas créé d'objet qui la maintienne en vie, `POE` va la passer au ramasse-miettes.

Le noyau va alors se rendre compte qu'il ne reste plus de tâches en cours, et va donc sortir de la méthode `run()`. Le programme s'arrêtera alors.

Envoyer un événement

```
POE::Kernel->yield
```

```
use POE;

POE::Session->create(
  inline_states => {

    _start => sub {
      POE::Kernel->yield("tick");
    },

    # définition de l'événement "tick"
    tick => sub {
      say scalar localtime;
      POE::Kernel->delay(tick => 1);
    },
  },
);

POE::Kernel->run;
```

Ici, la session lors de sa création (dans la fonction traitant l'événement `_start`) va s'envoyer un événement à elle-

même avec la méthode `yield()` du noyau. Le noyau sait donc que la session a du travail en cours, étant donné qu'il y a un événement qui lui est destiné. La session ne sera pas détruite tout de suite.

Le noyau va s'occuper d'envoyer l'événement et d'appeler le callback associé. Celui-ci est défini en ajoutant un événement auquel devra réagir la session dans les `inline_states`, avec le callback associé.

Le callback est donc appelé par le noyau, permettant d'imprimer la date et l'heure courante. Le gestionnaire d'événements va alors à nouveau s'envoyer un événement `tick`, mais au bout d'une seconde.

Ceci se fait avec la méthode `delay()` du noyau. En effet, si la fonction Perl `sleep` avait été utilisée, le programme se serait mis en pause... et le noyau aussi ! Et les autres sessions, s'il y en avait, n'auraient pas été planifiées tout de suite, ce qui aurait ralenti tout le programme – là où le but était de mettre en pause la session courante seulement !

Bien sûr, le fait de planifier l'envoi d'un événement indique au noyau de garder la session courante active, pour pouvoir traiter les événements futurs.

Le programme ci-dessus va donc imprimer la date et l'heure courantes toutes les secondes, sans jamais s'arrêter.

Passer des paramètres

Passer un argument se fait très simplement à la suite du nom d'événement à planifier :

```
use POE;

POE::Session->create(
  inline_states => {
```

```
    _start => sub {
      # passage d'argument
      POE::Kernel->yield( next => 10 );
    },

    next => sub {
      my $i = $_[ARG0];
      say $i;
      POE::Kernel->yield( next => $i-1 )
        unless $i == 0;
    },
  },
);

POE::Kernel->run;
```

Pour le récupérer, toutefois, il faut le chercher à un offset particulier de `$@_`.

Info

POE passe un grand nombre de paramètres additionnels aux callbacks (dont la session, la pile, l'objet courant, une référence sur le noyau lui-même, etc.). POE exporte donc des constantes permettant de récupérer facilement les paramètres souhaités.

Le premier argument passé lors de l'appel à `yield()` sera accessible *via* `$_[ARG0]`, le suivant *via* `$_[ARG1]`, jusqu'au dernier disponible *via* `$_[$#_]`. Les paramètres sont en effet ajoutés en dernier, donc tous les arguments peuvent être récupérés ainsi :

```
my @params = @_[ARG0 .. $#_];
```

Dans l'exemple ci-dessus, le callback va récupérer la valeur du paramètre dans `$i`, l'afficher, puis lancer un autre événement en décrémentant le paramètre. Si le paramètre `$i` vaut 0, l'événement n'est pas planifié – et donc la ses-

sion ne sert plus à rien, passe donc au ramasse-miettes, et le programme s'arrête.

Le programme va donc afficher les valeurs 10 à 0, puis s'arrêter.

Utiliser des variables privées

$_[HEAP]

Une session dispose d'un espace réservé – la pile (*heap*) – lui permettant de stocker des données. Ces données sont privées et accessibles uniquement par la session. En pratique, la pile est simplement un scalaire, qui est la plupart du temps une référence de hash pour stocker plusieurs variables :

```
use POE;

foreach my $counter ( 1, 3 ) {
  POE::Session->create(
    args          => [ $counter ],
    inline_states => {

      _start => sub {
        my ($h, $i) = @_[HEAP, ARG0];
        $h->{counter} = $i;
        POE::Kernel->yield("next");
      },

      next => sub {
        my $h = $_[HEAP];
        say $h->{counter};
        POE::Kernel->yield("next")
          unless $h->{counter}-- == 0;
      },
    },
```

```
    );
}

POE::Kernel->run;
```

Astuce

Notez le paramètre `args` de la méthode `create()`, spécifiant une liste à passer en argument lors de l'événement `_start`. Ici, un seul argument est passé, mais l'usage d'une référence de liste permet d'en passer autant que nécessaire.

Dans cet exemple, deux sessions seront lancées et décrémenteront leurs paramètres jusqu'à zéro, à tour de rôle. La sortie du programme sera donc :

```
1  # première session
3  # deuxième session
0  # première, qui s'arrête
2  # deuxième à nouveau
1  # deuxième
0  # deuxième, qui s'arrête
   # le programme s'arrête aussi
```

Nous commençons à voir l'intérêt de `POE`, qui va donc empiler les événements générés et les délivrer au fur et à mesure aux sessions concernées. Le noyau traite les événements avec une politique « premier arrivé, premier servi » ou FIFO (*First In, First Out*).

Communiquer avec une autre session

```
POE::Kernel->post
```

Jusqu'à présent, les exemples montraient des sessions qui s'envoyaient des événements à elles-mêmes. Mais bien sûr `POE` permet aux sessions de communiquer entre elles et de se transmettre des informations.

Dans l'exemple qui suit, une première session attend des événements `debug` pour les afficher. Une deuxième session va décrémenter un compteur, en envoyant des messages à la première pour indiquer ce qu'il se passe[2] :

```
use POE;

POE::Session->create(
  inline_states => {

    _start => sub {
      POE::Kernel->alias_set("logger");
    },

    debug => sub {
      my @args = @_[ARG0 .. $#_];
      say @args;
    },
  },
);

POE::Session->create(
  inline_states => {
      _start => sub {
      POE::Kernel->yield( tick => 3 );
    },

    tick => sub {
      my $i = $_[ARG0];
```

2. Même si l'action de log est ici très simple, cela permettrait par exemple d'avoir une session connectée à une base de données, ou faisant des traitements complexes (en multiplexant les logs sur différents supports par exemple). L'exemple n'est donc pas si tiré par les cheveux que cela...

```
        # passage de message à la 1ère session
        POE::Kernel->post( logger => debug
    ➥       => "test - i vaut $i" );
        POE::Kernel->yield(tick => $i-1)
          unless $i == 0;
      },
    },
);

POE::Kernel->run;
```

Si rien n'était fait lors de son démarrage, la première session créée par `POE::Session->create` démarrerait, mais s'arrêterait immédiatement. En effet, comme elle n'aurait rien à faire, `POE` la passerait au ramasse-miettes.

L'astuce consiste donc à lui affecter un `alias`, qui est un nom symbolique pour la session. Comme tout le monde peut envoyer un message à un alias, `POE` ne peut savoir à l'avance si la session est utile ou pas. Dans le doute, il va donc garder la session en vie.

Et heureusement, car la deuxième session va lui envoyer des messages avec la ligne :

```
POE::Kernel->post( logger => debug
    ➥   => "test - i vaut $i" );
```

La syntaxe est assez simple : la méthode `post()` du noyau permet de spécifier dans l'ordre le destinataire, l'événement à envoyer et des paramètres éventuels.

Une fois que la deuxième session a envoyé ses quatre messages, elle va s'arrêter. Mais `POE` n'arrête toujours pas la première session, car elle a un alias ! Ce qui pose problème, car le programme dans ce cas ne s'arrêterait jamais, sans toutefois ne rien faire. En effet, la première session serait toujours en attente, mais aucune autre session n'existant, elle attendrait en vain. . .

Pour empêcher cela, POE va détecter quand il ne reste que des sessions maintenues en vie grâce à des alias. Si c'est le cas, le noyau va envoyer un signal IDLE à toutes les sessions, pour leur donner une chance de se réveiller et de recommencer à travailler. Si aucune ne recommence à travailler, alors le noyau enverra un signal ZOMBIE qui n'est pas gérable par les sessions : celles-ci vont donc mourir et le programme se terminer... Ouf !

Attention

Un programme POE implique une bonne coopération des sessions entre elles. En effet, si l'un des gestionnaires d'événements bloque pour une raison ou pour une autre, c'est l'ensemble du programme qui s'arrête.

Les sections suivantes indiquent donc quelques problèmes potentiels et la manière d'y remédier pour que le programme POE continue à bien fonctionner.

Envoyer un message différé

POE::Kernel->delay

Ce point a déjà été abordé dans un exemple (voir page 333), mais il est bon de le rappeler : la fonction sleep() est à bannir. Avec delay, la session indique au noyau POE qu'elle souhaite recevoir un événement après un certain nombre de secondes. Elle peut aussi s'envoyer des paramètres supplémentaires :

```
# s'envoyer un événement différé
# dans $sec secondes
POE::Kernel->delay(event => $sec, @params);
```

Les machines disposant de `Time::HiRes` (c'est-à-dire tous les Perl depuis la version 5.8) peuvent utiliser des fractions de secondes. Cependant, Perl n'est pas un langage dit temps réel et la précision de l'alarme dépend de beaucoup de choses, dont le système d'exploitation.

Il n'est pas possible d'envoyer un événement différé à une autre session. Pour cela, il faut s'envoyer un événement différé, qui se chargera d'envoyer le message à la deuxième session :

```
POE::Session->create(
  inline_states => {
    _start => sub {
      POE::Kernel->delay(later => 5);
    },
    later => sub {
      POE::Kernel->post(session => "event");
    },
);
```

Envoyer un message à l'heure dite

`POE::Kernel->alarm`

Quand le programme doit se réveiller à une date et une heure précises, il ne faut plus utiliser `delay` mais la fonction `alarm`.

```
# s'envoyer un événement à la date/heure
# indiquée par $epoch
POE::Kernel->alarm(event => $epoch,@params);
```

La date est fournie au format Epoch[3].

Terminer le programme

Pour donner une chance à toutes les sessions de se terminer correctement, il faut bien sûr bannir les appels à `exit` ou `die`. En effet, ces fonctions terminent le programme d'un seul coup – sans laisser aux sessions un délai de grâce.

Le bon citoyen `POE` enverra des messages de fin aux diverses sessions :

```
# il est temps de s'arrêter !
POE::Kernel->post($_ => "shutdown")
  for @sessions;
```

Charge à elles de réagir à ce signal en s'arrêtant proprement : fermeture des fichiers et connexions, suppression des alias de session, arrêt des composants. La session sera alors passée au ramasse-miettes, et quand il ne restera plus aucune session, le programme s'arrêtera.

Couper les longs traitements

Certains événements nécessitent un long traitement, par exemple, en lançant une boucle itérant un grand nombre de fois.

Mais du fait de sa nature monoprocessus et monothread, l'ensemble du programme `POE` va se retrouver bloqué le temps que le traitement de l'événement se termine. Ceci peut être inacceptable, car pendant ce temps, aucun événement n'est acheminé – et donc aucun callback appelé.

3. Nombre de secondes depuis la *date initiale* (qui dépend du système d'exploitation, souvent le 1er janvier 1970).

Imaginez une interface graphique avec une telle boucle : l'utilisateur aura beau cliquer, rien ne se passera... jusqu'à ce que le traitement abusif soit terminé, et alors toutes les interactions seront traitées d'un coup ! Pire encore, une application réseau mal codée utilisant un très long callback pourra bien voir toutes les connexions en cours perdues, pour cause de réponse trop lente de l'application !

L'exemple suivant calcule la somme de 0 à 1 000 000, tout en affichant la date et l'heure toutes les secondes. Pour ne pas perturber l'affichage[4], la boucle va être découpée pour ne faire qu'une itération par événement :

```
use POE;

POE::Session->create(
  inline_states => {

    _start => sub {
      POE::Kernel->yield("tick");
      POE::Kernel->yield("sum");
    },

    tick => sub {
      say scalar localtime;
      POE::Kernel->delay(tick => 1);
    },

    sum => sub {
      state $i   = 0;
      state $sum = 0;
      $sum += $i++;
      if ( $i > 1_000_000 ) {
        # affichage du résultat final
        say "sum = $sum";
```

4. Cet exemple est naïf, mais imaginez un traitement complexe à chaque itération...

```
        # arrêt de l'horloge
        POE::Kernel->delay("tick");
      }
      else {
        POE::Kernel->yield("sum");
      }
    },

  },
);

POE::Kernel->run;
```

Il y a plusieurs lignes intéressantes dans cet exemple.

Les variables définies avec le mot-clé `state` gardent leur valeur d'une invocation du callback à la suivante. Cependant, si plus d'une session était créée, ces variables seraient partagées et modifiées par les deux sessions. Ce n'est généralement pas le but recherché. Pour pallier cet inconvénient, il faut alors utiliser la pile qui est, elle, privée à la session.

Il est aussi intéressant de noter la ligne :

```
# arrêt de l'horloge
POE::Kernel->delay("tick");
```

L'appel de `delay` avec seulement le nom de l'événement va annuler la planification de cet événement déjà réalisée *via* un appel à `delay`.

Malheureusement, même s'il permet de laisser aux autres événements le temps de s'exécuter, le fait de s'envoyer un million de messages (un par itération) est très inefficace. Là où une simple boucle :

```
perl -E '$s+=$_ for 0..1_000_000; say $s'
```

se termine en moins d'une seconde, le programme basé sur `POE` ci-dessus met... un peu plus d'une minute ! Et même si l'exemple est vraiment trivial et non représentatif de ce qui peut se faire dans le monde réel, cela n'en reste pas moins inacceptable.

L'astuce consiste donc à découper le long traitement non pas en très fines tranches, mais en tranches assez conséquentes pour que le surcoût ne soit pas trop gros, mais assez fines pour ne pas perturber le flux du programme. Dans l'exemple pris, les itérations ne se feront donc pas une par une (trop lent) ou par un million (pas assez réactif), mais mille par mille[5]. La boucle du programme devient donc :

```
sum => sub {
  state $i   = 0;
  state $sum = 0;
  $sum += ( $i > 1_000_000 ? 0 : $i++ )
    for 0..999;

  if ( $i > 1_000_000 ) {
    # affichage du résultat final
    say "sum = $sum";
    # arrêt de l'horloge
    POE::Kernel->delay("tick");
  }
  else {
    POE::Kernel->yield("sum");
  }
},
```

Et ainsi, le temps d'exécution revient en dessous de la seconde...

5. Ce chiffre dépend bien sûr du programme et de la complexité de chaque itération du traitement.

Attention

Notez le test lors de l'addition : comme la boucle est tronçonnée, les tranches ne sont peut-être pas un multiple entier de la boucle totale. Il faut donc s'assurer que cette astuce ne fausse pas le résultat du programme : mieux vaut un programme lent qu'un programme faux !

Bannir les entrées-sorties bloquantes

Un autre point qui peut poser problème dans l'environnement POE concerne les entrées-sorties bloquantes. Pour éviter que le programme ne s'arrête en attente d'une entrée-sortie quelconque, il faut s'assurer que les connexions sont ouvertes en mode non bloquant, ne pas s'arrêter lors de la lecture des fichiers ou pipes, ne pas bloquer en attente de données non présentes... Heureusement, POE propose des solutions pour simplifier la tâche du programmeur.

Réutilisation de code

POE n'est pas seulement un framework événementiel élaboré, c'est aussi tout un écosystème de modules fournissant des composants réutilisables facilement dans tout programme POE.

Ces modules sont répartis en trois grandes catégories, suivant des degrés d'abstraction variables, et fournissant une balance différente entre les efforts de développement et le contrôle qu'aura le développeur sur le comportement de ces modules.

Il est bien sûr possible de mélanger des composants de différents niveaux, permettant ainsi une flexibilité accrue.

Composants de haut niveau

POE::Component

Les « composants » (*component* en anglais) sont puissants et fournissent des services avancés à peu de frais. Souvent une ou plusieurs sessions `POE` distinctes seront créées, qui dialogueront avec les autres sessions comme un composant distinct du programme. Ces composants se retrouvent dans l'espace de noms `POE::Component`.

Une grande partie d'entre eux propose par exemple des clients ou des serveurs réseau : accès ou fourniture de services web, syslog, résolution de nom sont disponibles directement, mais aussi les briques nécessaires pour créer son propre client ou serveur générique. Le programme suivant implémente un serveur d'*echo* complet :

```
use POE;
use POE::Component::Server::TCP;

POE::Component::Server::TCP->new(
  Port         => 17000,
  ClientInput  => sub {
    my ($heap, $input) = @_[HEAP, ARG0];
    $heap->{client}->put($input);
  },
);

POE::Kernel->run;
```

Parmi les autres principaux composants de haut niveau se trouvent des interfaces à des bibliothèques ou des applications : `libpcap`, `oggenc`, `mpd`, etc. CPAN regroupe environ 600 composants de haut niveau pour `POE`.

Boîte à outils de niveau intermédiaire

POE::Wheel

Lorsque les « POCO[6] » ne sont pas appropriés, ou trop spécialisés, il faut réutiliser des « roues » standard de plus bas niveau.

Ces « roues » fournies par les modules de l'espace de noms `POE::Wheel` (d'où leur nom) sont plus généralistes. Ce sont les briques de base de nombreux composants.

Elles ne créent pas de nouvelles sessions, mais ajoutent des traitements d'événements à la session utilisant la roue – autrement dit, la session est modifiée. Comme les roues ne créent pas de session autonome, le noyau `POE` ne les stockera pas : c'est à la session qui utilise la roue de la stocker... et de la détruire quand elle ne sert plus (pas de ramasse-miettes automatique fourni par le noyau). Les roues sont donc étroitement couplées avec leur session mère, et ne peuvent être passées ou utilisées par d'autres sessions : celles-ci doivent créer leur propre instance de la dite roue.

Parmi les roues disponibles, peuvent être citées la surveillance des lignes d'un fichier (à la `tail -f`), la gestion d'entrées-sorties non bloquantes (connexions réseau comprises), le lancement et le contrôle de processus, etc.

Leur utilisation implique bien sûr plus d'efforts de programmation qu'un composant fournissant un service clé en main, mais c'est le prix à payer pour plus de contrôle.

6. « POCO » comme `POE::Component`, les composants de haut niveau vus ci-dessus.

Fonctions de bas niveau POE

Finalement, quand même les roues fournies par `POE` sont trop évoluées, `POE` propose ses propres fonctions de bas niveau. Le code résultant sera certes moins raffiné et plus verbeux, mais fournira le contrôle le plus complet aux développeurs.

Les fonctions proposées permettent de gérer les alarmes et les signaux (au sens Unix du terme) et des interfaces aux appels système tels que `select`.

Bien sûr, l'ensemble des fonctionnalités d'orchestration et de passage de messages fournies par `POE` restent disponibles...

Astuce

Les différents niveaux des composants `POE` permettent aux développeurs de choisir le degré d'abstraction qu'ils souhaitent mettre en œuvre dans leur programme. Le nombre de briques de base disponibles donne la possibilité d'écrire un programme complexe en assemblant des composants éprouvés, et cela participe aussi de l'intérêt d'utiliser `POE`.

Exemple d'utilisation : le composant `DBI`

```
POE::Component::EasyDBI
```

Voici une illustration du fait que la programmation événementielle impose de s'habituer à tout penser de manière asynchrone. L'accès aux bases de données est un exemple typique de service dont l'API est fondamentalement synchrone, et qui induit donc des temps de latence. Le prin-

cipe général utilisé pour fournir un composant asynchrone à de tels services est de déporter la synchronicité dans un processus séparé avec lequel le composant POE discute au travers d'un tube et de POE::Wheel::Run.

Il existe quelques composants de pilotage DBI pour POE sur CPAN, tous fonctionnant sur le même principe général : pour chaque connexion à une base de donnée, le composant crée un nouveau processus qui se charge d'exécuter les commandes DBI.

Le composant qui va être présenté plus en détails est POE::Component::EasyDBI, parce qu'il est un peu plus agréable à manipuler que les autres, et parce que c'est celui utilisé par l'un des auteurs. Fondamentalement, tous les composants d'accès aux bases de données reposent sur le même principe de découpage des requêtes en deux événements POE : le premier pour préparer la requête et passer les éventuels arguments, et le second pour la réception des résultats.

Il faut d'abord créer une session POE::Component::EasyDBI qui se charge de la communication avec la base de données. Les paramètres passés à la méthode spawn() seront familiers à l'utilisateur averti de DBI :

```
POE::Component::EasyDBI->spawn(
  alias     => "events_dbh",
  dsn       => "dbi:mysql:database=events",
  username  => "user",
  password  => "pass",
);
```

spawn() accepte bien sûr d'autres options, mais les réglages par défaut conviennent en général très bien. Le nom de la session, si aucun alias n'est indiqué, est "EasyDBI".

EasyDBI propose un grand nombre de commandes, chacune spécialisée dans un usage donné, un peu comme les nombreuses méthodes de DBI.

La commande `insert` sert pour exécuter des requêtes `INSERT` :

```
sub store_event {
  my ($kernel, $host, $service, $id, $msg)
  ➥ = @_[ KERNEL, ARG0 .. $#_ ];

  my $sql = q{
    INSERT INTO events
           (host, service, id, message)
    VALUES ( ?, ?, ?, ? )
  };

  $kernel->post(
    $easydbi,
    insert => {
      sql    => $sql,
      event => "store_event_result",
      placeholders
        => [$host, $service, $id, $msg],
    },
  );
}
```

Les paramètres sont assez simples à comprendre : `sql` pour donner la requête SQL, `placeholders` pour les éventuelles valeurs à passer en arguments de la requête et `event` pour indiquer le nom de l'événement de la session courante qui sera appelé pour recevoir les résultats. Celui-ci reçoit une référence vers un hash qui contient des champs différents en fonction de la commande exécutée, ainsi que quelques champs communs à toutes : `sql` contient la requête SQL envoyée, `placeholders` la référence vers les éventuels valeurs, `result` le résultat si la commande a réussi et `error` la chaîne d'erreur si elle a au contraire échoué. C'est d'ailleurs le premier champ à vérifier dans les fonctions de traitement :

```
sub store_event_result {
  my ($kernel, $db_result)
  ➥ = @_[ KERNEL, ARG0 ];
  warn "store_event: $db_result->{error}\n"
    if $db_result->{error};
}
```

La commande `do` sert pour la plupart des requêtes SQL hors `SELECT`, comme `UPDATE`, `DELETE`, etc.

```
sub delete_event {
  my ($kernel, $host, $service, $id, $msg)
  ➥ = @_[ KERNEL, ARG0 .. $#_ ];

  my $sql = q{
    DELETE
    FROM    events
    WHERE   host    = ?
        AND service = ?
        AND id      = ?
  };

  $kernel->post(
    $easy_dbi,
    array => {
      sql   => $sql,
      event => "delete_event_result",
      placeholders
        => [ $host, $service, $id ],
    },
  );
}

sub delete_event_result {
  my ($kernel, $db_result) =
  ➥ @_[ KERNEL, ARG0 ];
  warn "delete_event: $db_result->{error}\n"
    if $db_result->{error};
}
```

Plusieurs commandes existent pour les requêtes `SELECT`, de manière similaire aux différentes méthodes de DBI, pour récupérer les données sous des formes et des structures variées : `single` pour une valeur unique (comme `fetchrow_array()`), `hash` pour un ligne de plusieurs colonnes nommées (`fetchrow_hashref()`), `array` pour plusieurs lignes d'une colonne (avec jointure de colonnes si nécessaire), `arrayarray` pour plusieurs lignes de plusieurs colonnes (`fetchall_arrayref()`), `hashhash` pour plusieurs lignes de plusieurs colonnes nommées (comme `fetchall_hashref()`), `arrayhash` pour plusieurs lignes colonnes nommées (mais sous la forme d'un tableau de `fetchrow_hashref()`).

Voici un exemple qui récupère une simple valeur :

```
sub fetch_event {
  my ($kernel, $host) = @_[ KERNEL, ARG0 ];

  my $sql = q{
      SELECT  count(*)
      FROM    events
      WHERE   host = ?
  };

  $kernel->post(
    $easydbi,
    single => {
      sql   => $sql,
      event => "fetch_event_result",
      placeholders => [ $host ],
    },
  );
}

sub fetch_event_result {
  my ($kernel, $db_result)
    ➥    = @_[ KERNEL, ARG0 ];
```

```
  if ($db_result->{error}) {
    warn "fetch_event: $db_result->{error}
    ➥ \n";
    return;
  }

  print $db_result->{result},
    " événements associés à l'hôte ",
    "$placeholders->[0]\n";
}
```

Et un exemple qui récupère les résultats sous forme d'un tableau de structures :

```
sub find_events {
  my ($kernel, $host) = @_[ KERNEL, ARG0 ];

  my $sql = q{
    SELECT  *
    FROM    events
    WHERE   host like ?
  };

  $kernel->post(
    $easydbi,
    arrayhash => {
      sql    => $sql,
      event  => "find_events_result",
      placeholders => [ $host ],
    },
  );
}

sub find_events_result {
  my ($kernel, $db_result)
    = @_[ KERNEL, ARG0 ];
  if ($db_result->{error}) {
    warn "find_events: $db_result->{error}
    ➥ \n";
    return;
  }
```

```
  for my $event (@{$db_result->{result}}) {
   print "traitement de l'événement pour /",
          $event->{host},    "/",
          $event->{service}, "/",
          $event->{id},      "\n";

   $kernel->yield("process_event",
 ➥  $event);
  }
}
```

`POE::Component::EasyDBI` permet de passer des données supplémentaires aux événements traitant les résultats : il suffit d'ajouter les champs correspondants dans le hash passé à la commande `EasyDBI`, qui se retrouveront dans le hash fournit en résultat. De manière assez évidente, il vaut donc mieux utiliser des noms de champs qui soient propres, par exemple en les préfixant par des doubles espaces soulignées.

Petit exemple pour mesurer la latence au niveau base de données :

```
sub store_event {
  my ($kernel, $host, $service, $id, $msg)
    ➥    = @_[ KERNEL, ARG0 .. $#_ ];

  my $now_ms = gettimeofday();

  my $sql = q{
    INSERT INTO events
           (host, service, id, message)
    VALUES ( ?, ?, ?, ? )
  };

  $kernel->post(
      $easydbi,
      insert => {
      sql    => $sql,
      event => "store_event_result",
      __start_time => $now_ms,
```

```
        placeholders
          => [$host, $service, $id, $msg],
      },
  );
}

sub store_event_result {
  my ($kernel, $db_result)
    ➥ = @_[ KERNEL, ARG0 ];

  warn "store_event: $db_result->{error}\n"
      if $db_result->{error};

  my $now_ms = gettimeofday();
  printf "La requête SQL a mis %.2f sec\n",
    ➥ $now_ms-$db_result->{__start_time};
}
```

Bien sûr, `POE::Component::EasyDBI` n'a pas été présenté en profondeur ici, mais l'important était de signaler son existence, et l'intérêt qu'il y a à utiliser ce genre de composants. Il est clair qu'il induit un surcoût non négligeable en termes de nombre de fonctions ou événements `POE` à écrire, puisque pratiquement chaque requête SQL implique deux événements. Néanmoins, le gain que cela apporte en termes de bande passante d'exécution au niveau du noyau `POE` est très sensible et en justifie le coût.

Intégration à Moose

Il est possible de reprocher à `POE` sa verbosité, en particulier lors de l'emploi d'une pile. Heureusement, le module `MooseX::POE` permet de marier élégamment `POE` avec tous les avantages de `Moose`. L'exemple rencontré auparavant montrant comment avoir des variables privées peut donc se réécrire de la sorte :

```
package Counter;
use MooseX::POE;

# attribut propre à chaque session
has count => (
  is => "rw",
  isa => "Int",
  required => 1
);
sub START {
    my $self = shift;
    $self->yield("next");
}
event next => sub {
    my $self = shift;
    my $i = $self->count;
    say $i;
    $self->count($i-1);
    $self->yield("next")
      unless $i == 0;
};
Counter->new(count=>3);
Counter->new(count=>1);
POE::Kernel->run;
```

Chaque session sera définie dans une classe, et créée en instanciant cette classe. La méthode `START` sera automatiquement appelée lors du démarrage de la session, après la création de l'objet lui-même. Les événements seront définis avec le mot-clé `event` prenant en compte deux paramètres : le nom de l'événement et le callback associé. Bien sûr, tous les avantages de `POE` (définition d'attributs, vérifications diverses, etc.) se retrouvent dans `MooseX::POE`, avec en plus quelques méthodes d'aide telles que `yield()`, faisant exactement la même chose que la méthode du noyau du même nom.

L'utilisation de la notation objet avec `MooseX::POE` permet donc d'avoir un code plus épuré, plus lisible… et donc plus maintenable.

19

POE distribué

Bien que les avantages de POE soient nombreux, son problème principal découle de l'un d'entre eux, à savoir sa nature monoprocessus et monothreadée. En effet, si ce mode de fonctionnement aide énormément l'écriture de tout programme, il ne facilite pas le passage à l'échelle. Les performances seront donc limitées à la puissance brute d'un seul processeur ; ajouter des cœurs ou d'autres processeurs ne rendra pas le programme plus rapide...

Pour pallier ce problème, le composant[1] POE IKC (*Inter Kernel Communication*) permet à plusieurs programmes POE de se passer des messages et des événements comme s'ils étaient orchestrés par le même noyau POE.

Les avantages sont évidents : faire tourner plusieurs programmes permet au système d'exploitation de les orchestrer sur autant de processeurs et de cœurs que disponibles. Mieux même, IKC permet de faire communiquer de manière transparente des programmes POE sur différentes machines, à travers le réseau !

1. Au sens « POCO ».

Créer un serveur IKC

```
use POE::Component::IKC::Server
```

L'illustration d'un serveur IKC se fait en reprenant l'exemple du logger, mais qui sera cette fois disponible en tant que service. Il tournera dans un serveur distinct, accessible sur le port 17000 :

```
use POE;
use POE::Component::IKC::Server;

# création du serveur IKC
POE::Component::IKC::Server->spawn(
  name => "my_server",
  ip   => "localhost",
  port => 17000,
);

POE::Session->create(
  inline_states => {

    _start => sub {
      POE::Kernel->alias_set("logger");
      POE::Kernel->post( IKC
        => publish
        => logger
        => [ "debug" ]
      );
    },

    debug => sub {
      my @args = @_[ARG0 .. $#_];
      say @args;
    },
  },
);

POE::Kernel->run;
```

Le programme va donc lancer un composant serveur IKC (avec la méthode `POE::Component::IKC::Server->spawn`), en lui donnant un nom, ainsi que les paramètres réseau sur lesquels écouter. Une session va alors être créée.

Lors de son démarrage, cette session va prendre un alias ; mais surtout elle va s'enregistrer auprès de IKC en listant la session et les événements qui seront accessibles par les clients du service. Cela se fait en envoyant le message `publish` à la session IKC.

La nouvelle session définit aussi l'événement `debug`, celui-là même qui est publié auprès de IKC. Comme dans l'exemple précédent, il ne fait qu'afficher les paramètres qui lui sont passés – mais il pourrait faire beaucoup plus...

Créer un client IKC

```
use POE::Component::IKC::Client
```

Le client n'est pas tellement plus compliqué :

```
use POE;
use POE::Component::IKC::Client;
use POE::Component::IKC::Responder;

POE::Component::IKC::Responder->spawn;
POE::Component::IKC::Client->spawn (
  ip          => "localhost",
  port        => 17000,
  name        => "my_client",

  on_connect => sub {
    POE::Session->create(
      inline_states => {
```

```
        _start => sub {
          POE::Kernel->yield("tick");
        },

        tick => sub {
          POE::Kernel->post( IKC
           => post
           => "poe://my_server/logger/debug"
           => scalar localtime
          );
          POE::Kernel->delay(tick => 1);
        },

      },
    );
  },
);

POE::Kernel->run;
```

Un composant `Responder` est créé (sans paramètre), il sera utilisé en interne par IKC qui en a besoin pour discuter avec le serveur. La création du client IKC lui-même se fait ainsi :

```
POE::Component::IKC::Client->spawn(%params);
```

Outre les paramètres réseau permettant de joindre le serveur, le client va prendre un nom et un callback. Ce callback sera appelé lorsque la connexion au serveur est effective. En effet, il n'est pas judicieux de vouloir utiliser le serveur tant qu'il n'est pas disponible...

Le callback va ici simplement créer une nouvelle session. Celle-ci va à nouveau s'envoyer des événements toutes les secondes, qui seront traités en utilisant le service `debug` proposé par le serveur. Cela se réalise en envoyant l'événement `post` à la session `IKC`, qui utilisera le premier pa-

ramètre pour savoir à quel serveur, session et événement envoyer le message.

La syntaxe pour décrire l'événement ciblé est :

```
poe://$server/$session/$event
```

En effet, il est tout à fait possible de se connecter à plusieurs serveurs IKC (un par service), ou qu'un serveur accepte plusieurs événements destinés à plusieurs sessions différentes.

IKC se chargera en interne de toute la communication réseau, sérialisera et désérialisera (avec `Data::Dumper`) automatiquement les données.

Attention

Un point important sur la sécurité : un utilisateur du service peut passer n'importe quoi à la session serveur, qui le désérialisera dans un `eval` : cela peut impliquer de l'exécution de code pour un client malveillant…

Aller plus loin

Cette introduction ne fait qu'effleurer le sujet de POE distribué. Beaucoup de choses peuvent être faites en utilisant IKC. Pour en citer quelques-unes :

- Passage d'événements retour : avec ou sans RSVP (retour différé dans le temps). Cela permet de créer ou utiliser des protocoles de type RPC, synchrones ou asynchrones.
- Communication *peer to peer* : avec un noyau IKC serveur sur lequel se connectent plusieurs client IKC de type « esclave » (*worker*). Les tâches sont ensuite planifiées par le serveur qui les distribue et collecte ensuite les résultats.

- Surveillance intégrée des noyaux distants : IKC fournit les moyens techniques pour suivre lorsqu'un noyau se déconnecte ou se (re-)connecte suite à une pertubation réseau (ou autres événements).

Bref, IKC permet le passage à l'échelle de POE, tant en termes de meilleure utilisation des performances que de modularisation des programmes. Vitesse et robustesse accrues, pour une maintenance facilitée : tels sont les avantages qu'IKC apporte à POE.

20

Analyse de documents HTML

Cette partie présente les moyens d'interagir en Perl avec le *World Wide Web*. Exactement comme un utilisateur aux commandes de son navigateur web, sauf que dans le cas présent, le client web (le « navigateur ») sera écrit en Perl.

Info

Le Web est la combinaison de trois éléments importants : le format HTML pour encoder les documents, le protocole HTTP pour les transférer et les URI pour les localiser. À partir d'une URI, un client saura à quel serveur demander le document correspondant, et utilisera HTML (si le document est encodé ainsi) pour l'analyser et pouvoir l'afficher.

Il existe de nombreuses manières d'analyser un document HTML. Ce chapitre présente très rapidement une approche simple mais déconseillée, avant de proposer trois modules qui permettent une analyse correcte du format HTML. Ces modules seront présentés en partant de celui de plus bas niveau, qui sert de base aux suivants. Les exemples utilisés seront les mêmes, afin de pouvoir comparer les différences entre les modules.

Analyser avec les expressions régulières

```
m{<b>(.*?)</b>}i
```

À première vue, un document HTML, c'est avant tout du texte. Et Perl dispose d'un outil surpuissant pour l'analyse de texte : *les expressions régulières*. Pourtant, *n'analysez jamais* des documents HTML avec des expressions régulières !

Utiliser des expressions régulières semble simple et rapide, mais (en dehors de scripts jetables qui ne seront jamais réutilisés), cette approche n'aura aucune pérennité à long terme.

Voici quelques-uns des inconvénients de l'utilisation des *regexp* dans le cadre de l'analyse de documents HTML :

- **Fragilité.** Il suffit qu'un caractère change dans le document à analyser pour que l'expression ne corresponde plus et renvoie n'importe quoi (ou rien du tout).
- **Inadaptation.** Elles ne sont pas adaptées à l'analyse d'éléments imbriqués (par exemple des listes de listes, avec un niveau arbitraire d'imbrication).
- **Mauvaise représentation.** HTML est un format qui représente des données structurées en arbre. Il y a une infinité de manières de représenter un contenu identique sémantiquement.

Les expressions régulières ne sont pas adaptées à la reconnaissance de données arborescentes, car elles analysent le texte à un très bas niveau : il faut tenir compte des blancs, des sauts de ligne, des guillemets simple ou double, des commentaires HTML, autant de choses qui peuvent va-

rier dans un document source, sans pour autant en changer la signification.

Utiliser l'analyseur événementiel HTML::Parser

La plupart des modules d'analyse HTML sont basés sur `HTML::Parser`. Ce module n'est pas un analyseur SGML générique (mais qui se souvient qu'HTML est défini en SGML[1] ?) ; il est capable d'analyser le HTML tel qu'il existe sur le Web, tout en disposant d'options permettant de respecter les spécifications du W3C (*World-Wide Web Consortium*) si cela est souhaité.

`HTML::Parser` est un analyseur HTML événementiel : l'analyse se fait à partir d'un flux de données, c'est-à-dire que le document pourra être reçu sous forme de morceaux de taille arbitraire par l'analyseur, et au fur et à mesure de l'analyse du document, des événements (voir page 369) sont déclenchés, et pris en charge par des fonctions de rappel (*callback*).

Instancier un analyseur HTML::Parser

```
HTML::Parser->new(. . .)
```

De manière classique, un analyseur `HTML::Parser` est un objet qui est créé grâce au constructeur `new` du module `HTML::Parser`.

1. SGML est un système permettant de définir des types de documents structurés et des langages de balisage pour représenter les instances de ces types de documents.

```
use HTML::Parser;
my $p = HTML::Parser->new(api_version => 3);
```

Lors de l'appel à `new()` ci-dessus, l'argument `api_version => 3` a été passé. Par défaut, un nouvel objet `HTML::Parser` est créé en utilisant la version 2 de l'API du module, qui suppose l'utilisation de noms de gestionnaires et de spécifications d'arguments prédéfinis. La version 3 de l'API est donc beaucoup plus souple.

La classe `HTML::Parser` est sous-classable. Les objets sont de simples tables de hachage, et `HTML::Parser` se réserve uniquement les clés commençant par `_hparser`. Cela permet d'utiliser l'objet `HTML::Parser` pour y stocker les états nécessaires au cours de l'analyse.

Créer un gestionnaire d'événements

```
$p->handler(. . .)
```

Les gestionnaires d'événement sont déclarés avec la méthode `handler()` :

```
$p->handler( event => \&handler, $argspec );
```

Dans l'exemple de code ci-dessus, `event` est le nom de l'événement à traiter, `\&handler` est une référence à la fonction codant le gestionnaire (il est également possible de donner simplement un nom de méthode) et `$argspec` est une chaîne de caractères représentant la spécification des arguments attendus par le gestionnaire (`handler()` accepte également d'autres arguments : la documentation du module en présente la liste complète).

Il est aussi possible de définir les gestionnaires lors de l'initialisation de l'analyseur, à l'aide de clés composées du nom de l'événement à traiter et du suffixe `_h` (*handler*).

```
$p = HTML::Parser->new(
  api_version => 3,
  event_h     => [ \&handler, $argspec ]
);
```

La valeur associée est une référence à un tableau contenant les mêmes paramètres que lors de l'appel à `handler()` (hormis le nom de l'événement).

`HTML::Parser` offre une grande souplesse pour la définition des gestionnaires d'événements : lors de leur déclaration (ce sont en général des fonctions spécifiques), il est possible de définir précisément quels paramètres ils vont recevoir.

Voici la liste des principaux paramètres (voir la documentation de `HTML::Parser` pour une liste complète) :

- `self` : référence à l'objet `HTML::Parser` ;
- `attr` : référence à une table de hachage des clés/valeurs des attributs ;
- `tagname` : le nom de la balise ;
- `text` : texte source (y compris les éléments de balisage) ;
- `dtext` : texte décodé (`&eacute;` sera transformé en « é »).

Voici un exemple qui montre la création d'un gestionnaire avec un événement. Pour la liste complète des événements disponibles, voir pages 370 et suivantes. Pour le détail sur la méthode `parse`, voir page 368.

```
use HTML::Parser;

my $p = HTML::Parser->new(api_version => 3);
$p->handler(
  start => \&handler, "tagname,text"
);
```

```
sub handler {
  my ($tagname, $text) = @_;
  say "détection de la balise $tagname";
  say "le source HTML : $text";
}

$p->parse("<html><body>Hello</body>
  ➥</html>");
```

Cet exemple produira sur la console :

```
détection de la balise html
le source HTML : <html>
détection de la balise body
le source HTML : <body>
```

Lancer l'analyse HTML

```
$p->parse(. . .)
$p->parse_file(. . .)
```

Une fois les gestionnaires définis, il reste à lancer l'analyse, au moyen d'une des méthodes suivantes :

- `$p->parse( $string )` : analyse le morceau suivant du document HTML. Les gestionnaires sont appelés au fur et à mesure. Cette fonction doit être appelée à répétition tant qu'il reste des données à analyser.
- `$p->parse( $coderef )` : il est également possible d'appeler `parse()` avec une référence de code, auquel cas cette routine sera appelée à répétition pour obtenir le prochain morceau de document à analyser. L'analyse se termine quand la référence de code renvoie une chaîne vide ou la valeur `undef`.

- `$p->parse_file( $file )` : réalise l'analyse directement depuis un fichier. Le paramètre peut être un nom de fichier, un *handle* de fichier ouvert, ou une référence à un *handle* de fichier ouvert.

  Si l'un des gestionnaires d'événement interrompt l'analyse par un appel à `eof()` avant la fin du fichier, alors `parse_file()` pourra ne pas avoir lu l'intégralité du fichier.

Terminer l'analyse du contenu

```
$p->eof()
```

Une fois toutes les données envoyées à l'analyseur, il est très important de signaler la fin de fichier, à l'aide de `$p->eof()`. Si l'un des gestionnaires termine prématurément l'analyse en appelant directement `$p->eof()`, la méthode `parse()` renvoie une valeur fausse. Dans le cas contraire, `parse()` renvoie une référence à l'analyseur lui-même, qui est une valeur vraie.

Détecter un nouveau document

```
start_document
```

Cet événement est déclenché avant tous les autres pour un nouveau document. Son gestionnaire pourra être utilisé à des fins d'initialisation. Il n'y a aucun texte du document associé à cet événement.

```
my $p = HTML::Parser->new(api_version => 3);

$p->handler(
  start => sub { say "début du document" },
);
```

Info

HTML::Parser gère neuf types d'événements, qui sont déclenchés dans différents cas.

Détecter une balise

```
declaration
```

Cet événement est déclenché quand une déclaration de balisage (*markup declaration*) est rencontrée. Dans la plupart des documents HTML, la seule déclaration sera `<!DOCTYPE . . .>`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
  Transitional//EN">
```

Détecter un commentaire

```
comment
```

Cet événement est déclenché quand un commentaire HTML est détecté.

```
use HTML::Parser;
```

```
my $p = HTML::Parser->new(api_version => 3);

$p->handler(
  comment => sub { say shift }, "tagname"
);

$p->parse(
  "<!-- ceci est un commentaire HTML -->"
);
```

Détecter un début de balise

```
start
```

Cet événement est déclenché lorsqu'une balise de début est reconnue.

```
use HTML::Parser;

my $p = HTML::Parser->new(api_version => 3);

$p->handler(
  start => sub { say shift }, "tagname"
);

$p->parse(
  '<a href="http://www.mongueurs.net/">'
);
```

Détecter du texte brut

```
text
```

Cet événement est déclenché lorsque du texte brut (des caractères) est détecté. Le texte peut contenir plusieurs lignes.

Une séquence de texte peut être découpée entre plusieurs événements `text` (sauf si l'option `unbroken_text` de l'analyseur est activée). Cependant, l'analyseur fait tout de même attention à ne pas couper un mot ou une série de blancs entre deux événements `text`.

```
use HTML::Parser;

my $p = HTML::Parser->new(api_version => 3);
$p->handler(
  text => sub { say shift }, "text"
);

$p->parse('<body>Du texte</body>');
```

Détecter la fin d'une balise

```
end
```

Cet événement est déclenché lorsqu'une balise de fin est reconnue, comme par exemple `</a>`.

Détecter la fin du document

```
end_document
```

Cet événement est déclenché lors de la fin du document. Il n'y a aucun texte du document associé à cet événement.

Détecter des instructions de traitement

```
process
```

Cet événement est déclenché lorsque des instructions de traitement (*processing instructions*) sont détectées. L'un des rares cas où cet événement sert est quand le document est du XHTML, qui contiendra au moins la balise `<?xml version="1.0"?>`

Capturer les autres événements

```
default
```

Cet événement est déclenché pour les événements qui n'ont pas de gestionnaire spécifique.

Extraire du texte d'un document

L'exemple suivant extrait le texte d'un document HTML. Le gestionnaire associé à l'événement `text` reçoit le paramètre `dtext` et en affiche directement le contenu.

```
use HTML::Parser;

# création de l'analyseur
my $p = HTML::Parser->new(api_version => 3);
$p->handler(
  text => sub { print $_[0] }, "dtext"
);

# analyse du document
$p->parse("");
$p->eof;
```

Cet analyseur est un peu naïf : d'une part, tous les blancs (espaces et sauts de lignes) seront affichés tels quels ; d'autre part, le contenu des blocs `<script>` et `<style>` sera également affiché.

Pour ignorer ces blocs, il faut configurer l'analyseur avec la méthode `ignore_elements()` :

```
$p->ignore_elements(qw(script style));
```

Pour supprimer les blancs inutiles, il faut modifier le gestionnaire :

```
sub { $_[0] =~ s/\s+/ /g; print "$_[0] "; }
```

On pourra également minimiser le nombre d'événements `text` reçus en configurant l'analyseur avec :

```
$p->unbroken_text;
```

Produire une table des matières

HTML est un format structuré, qui permet de définir des niveaux de titres dans un document (les balises `h1`, `h2`, jusqu'à `h6`). Cet exemple utilise les titres du document pour produire une table des matières.

```
# définition de l'analyseur
use HTML::Parser;

my $p = HTML::Parser->new(
  api_version => 3,
  start_h => [ \&start, "self, tagname" ],
  end_h   => [ \&end,   "self, tagname" ],
  text_h  => [ \&text,  "self, dtext"   ],
);

# gestionnaires d'événements
sub start { my ( $self, $tagname ) = @_;
  $tagname =~ /^h(\d+)$/
    and $self->{level} = $1;
}
```

```
sub end { my ( $self, $tagname ) = @_;
  if ( $tagname =~ /^h(\d+)$/ ) {
    my ( $level, $text )
      = delete @{$self}{qw( level text )};
    $text =~ s/\s+/ /g;
    $text =~ s/^\s+¦\s+$//g;
    say "  " x ( $level - 1 ), $text;
  }
}

sub text { my ( $self, $text ) = @_;
  $self->{level}
    and $self->{text} .= " " . $text;
}

# analyse du document
$p->parse($_) while <>;
$p->eof;
```

Info

Cet exemple est à stocker dans un fichier, ici `outline.pl`. Pour l'exécuter, il faut lui passer un nom de fichier HTML en paramètre. Il peut également s'utiliser en sortie de `pipe` sous unix.

```
perl ./outline.pl fichier.html
```

Dans cet exemple, le travail est réparti entre trois gestionnaires :

- `start` : se contente de tenir à jour le niveau de titre en cours (`level`).
- `text` : dans le cas où un niveau de titre est défini (`level` est vrai), concatène le texte décodé (argument `dtext`) dans la clé `text`.
- `end` : à la fermeture d'une balise de titre, affiche le texte accumulé jusque là, en éliminant les blancs surnuméraires.

Ce code s'appuie en particulier sur le fait qu'il ne peut y avoir de balises de titre imbriquées en HTML. Un document HTML invalide (par exemple `<h1>Titre 1<h2>Titre 2</h2></h1>`) provoquera des avertissements.

Voici ce que produit le programme sur le manuel de Gnuplot :

```
$ lwp-request http://www.gnuplot.info/docs/gnuplot.html ¦
     ./outline.pl

    Contents
Part I Gnuplot
    1 Copyright
    2 Introduction
    3 Seeking-assistance
    4 New features introduced in version 4.4
      4.1 Internationalization
      4.2 Transparency
      4.3 Volatile Data
      4.4 Canvas size
...
```

On constate en passant que cette documentation n'a pas de titres de niveau 2 et saute directement du niveau 1 au niveau 3.

Analyseur par token

L'analyse événementielle n'est pas toujours la plus intuitive. `HTML::TokeParser` est un analyseur par *token*, utilisable dans une programmation impérative. Les éléments analysés sont fournis par une méthode qui sera appelée à répétition, jusqu'à épuisement des données.

Le traitement ne se fait plus sur un flux, mais sur le document en entier. Il faut donc fournir un fichier, un descripteur de fichier ou une référence au texte complet du document au constructeur.

Créer une instance HTML::TokeParser

```
HTML::TokeParser->new()
```

Il y a plusieurs moyens d'initialiser `HTML::TokeParser`, et de créer un objet. L'extrait de code suivant présente les trois méthodes principales, à partir d'un fichier HTML, d'un descripteur de fichier, ou directement à partir d'une chaîne de caractères (`$html`).

```
use HTML::TokeParser;
# nom de fichier
my $p = HTML::TokeParser->new(
  $file, %options );

# descripteur de fichier
open my $fh, '<:utf8', 'index.html'
  or die "Erreur sur '$file' : $!";
my $p = HTML::TokeParser->new(
  $fh, %options );

# référence vers le texte du document
my $p = HTML::TokeParser->(
  \$html, %options );
```

Info

`HTML::TokeParser` est en réalité basé sur `HTML::Parser` en interne. C'est pourquoi les options éventuelles, sous forme d'une liste de clés-valeurs, sont les même que celles de `HTML::Parser`. Elles sont utilisées pour configurer l'analyseur HTML interne à `HTML::TokeParser`.

Récupérer des tokens

get_token

La principale méthode de `HTML::TokeParser` est `get_token`, qui renvoie le token suivant sous forme d'une référence de tableau. Le premier élément du tableau est une chaîne représentant le type de l'élément, et les paramètres suivants dépendent du type.

L'utilisation de `get_token` est très simple, il suffit de boucler :

```
use HTML::TokeParser;
use Data::Dumper;
my $html = '<html><body></body></html>';
my $p = HTML::TokeParser->(\$html);
while (my $token = $p->get_token()) {
  say Dumper($token);
}
```

Il y a six types de token, chacun ayant différents paramètres associés :

- Début de tag (S) :

```
["S", $tag, $attr, $attrseq, $text]
```

- Fin de tag (E) :

```
["E", $tag, $text]
```

- Texte (T) :

```
["T", $text, $is_data]
```

- Commentaire (C) :

```
["C",  $text]
```

- Déclaration (D) :

```
["D",  $text]
```

- Instruction de traitement (PI) :

```
["PI", $token0, $text]
```

La signification des tokens est la même que pour `HTML::Parser` (voir page 369).

`HTML::TokeParser` n'a pas de méthodes pour filtrer les balises, mais fournit plusieurs méthodes de plus haut niveau.

Obtenir des balises

```
get_tag( @tags )
```

Cette fonction renvoie la prochaine balise de début ou de fin, en ignorant les autres tokens, ou `undef` s'il n'y a plus de balises dans le document. Si un ou plusieurs argument sont fournis, les tokens sont ignorés jusqu'à ce que l'une des balises spécifiées soit trouvée.

Ce code trouvera la prochaine balise d'ouverture ou de fermeture de paragraphe :

```
$p->get_tag( "p", "/p" );
```

Le résultat est renvoyé sous la même forme que pour `get_token()`, avec le code de type en moins. Le nom des balises de fin est préfixé par un `/`.

Une balise de début sera renvoyée comme ceci :

```
[ $tag , $attr , $attrseq , $text ]
```

Et une balise de fin comme cela :

```
[ "/$tag", $text ]
```

Obtenir du texte

```
get_text( @endtags )
```

Cette fonction renvoie tout le texte trouvé à la position courante. Si le token suivant n'est pas du texte, renvoie une chaîne vide. Toutes les entités HTML sont converties vers les caractères correspondants.

Si un ou plusieurs paramètres sont fournis, alors tout le texte apparaissant avant la première des balises spécifiées est renvoyé.

Certaines balises peuvent être converties en texte (par exemple la balise `<img>`, grâce à son paramètre `alt`). Le comportement de l'analyseur est contrôlé par l'attribut `textify`, qui est une référence à un hash définissant comment certaines balises peuvent être converties. Chaque clé du hash définit une balise à convertir. La valeur associée est le nom du paramètre dont la valeur sera utilisée pour la conversion (par exemple, le paramètre `alt` pour la balise `img`). Si le paramètre est manquant, le nom de la balise en capitales entre crochets sera utilisé (par exemple `[IMG]`). La valeur peut également être une référence de code, auquel cas la fonction recevra en paramètre le contenu du token de début de balise, et la valeur de retour sera utilisée comme du texte.

Par défaut `textify` est défini comme :

```
{ img => 'alt', applet => 'alt' }
```

Obtenir du texte nettoyé

```
get_trimmed_text( @endtags )
```

Cette fonction opère comme `get_text()` mais remplace les blancs multiples par une seule espace. Les blancs en début et fin de chaîne sont également supprimés.

Extraire le texte d'un document avec HTML::Parser

Info

Dans les exemples qui suivent, l'analyseur lit le fichier donné sur la ligne de commande.

Les exemples sont les mêmes que dans les sections consacrées à `HTML::Parser`, afin de faciliter la comparaison entre les différents modules.

Voici une version naïve :

```
use HTML::TokeParser;
my $p = HTML::TokeParser->new(shift);

print $p->get_trimmed_text("/html");
```

Cette version a un petit défaut : le contenu des balises `<script>` et `<style>` est considéré comme du texte, et le code JavaScript ou CSS sera donc inclus dans le « texte » du document.

La version qui suit lit le texte du document jusqu'à rencontrer l'une des balises en question, saute à la balise de fin, et recommence jusqu'à épuisement du document :

```
use HTML::TokeParser;
my $p = HTML::TokeParser->new(shift);

# boucle sur la recherche de texte
my @skip = qw( script style );
while ( defined(
     my $t = $p->get_trimmed_text(@skip)
  ) ) {

  # affiche le texte obtenu
  print $t;

  # saute à la balise de fin ou termine
  # la lecture s'il n'y en a pas
  $p->get_tag(qw(/style /script)) or last;
}
```

Produire une table des matières avec HTML::Parser

Ici, seul le texte à l'intérieur des balises `<h1>` à `<h6>` doit être affiché. Les paramètres optionnels des méthodes `get_tag()` et `get_text()` seront bien utiles :

```
use HTML::TokeParser;
my $p = HTML::TokeParser->new(shift);

# définition de la liste des balises
my @start = qw( h1 h2 h3 h4 h5 h6 );
my @end = map { "/$_" } @start;
```

```
# boucle de recherche des débuts de titre
while (my $token = $p->get_tag(@start)) {

  # calcul du niveau de titre
  my ($level) = $token->[0] =~ /h(\d)/;

  # texte jusqu'à la fin de titre
  my $text = $p->get_trimmed_text(@end);

  # affiche le texte, avec indentation
  print "  " x ( $level - 1 ), $text, "\n"
      if $text;
}
```

Analyse par arbre avec HTML::TreeBuilder

Comme on l'a vu, HTML définit une structure arborescente. Les analyseurs présentés jusqu'ici n'utilisaient pas cet aspect fondamental, se contentant d'événements déclenchés ou de tokens détectés au fil de la lecture du document.

`HTML::TreeBuilder` est un module qui hérite de `HTML::Parser` pour construire une structure de données en arbre. `HTML::TreeBuilder` hérite également de `HTML::Element`. L'objet `HTML::TreeBuilder` est à la fois analyseur HTML et nœud racine d'un arbre composé d'autres objets de la classe `HTML::Element`.

Les méthodes issues de `HTML::Parser` sont utilisées pour construire l'arbre HTML, et celles issues de `HTML::Element` sont utilisées pour inspecter l'arbre.

(La page de manuel `HTML::Tree` pointe vers plusieurs pages de manuels et articles permettant de mieux comprendre la documentation de `HTML::TreeBuilder` et `HTML::Element`.)

Astuce

Il faut noter que le HTML est en général beaucoup plus difficile à analyser que ceux qui l'écrivent ne le croient. Les auteurs de `HTML::TreeBuilder` ont pris soin, pour produire une représentation arborescente correcte (cela n'est pas nécessaire quand on se contente de déclencher des événements ou produire un flux de tokens), de gérer un certain nombre de cas particuliers comme les éléments et balises de fin implicites.

Un exemple permet de mieux comprendre le problème, et la solution apportée par `HTML::TreeBuilder`. Soit le document HTML (complet !) :

```
<li>item
```

Le code suivant analyse le document et produit une sortie HTML correspondant à l'arbre construit (en ajoutant les balises implicites, telles que `<html>`, `<head>`, etc.) :

```
print HTML::TreeBuilder->new_from_content
    (<>)->as_HTML;
```

Le résultat est :

```
<html><head></head><body><ul><li>item </ul></body></html>
```

La sortie « XML » (en utilisant la méthode `as_XML()`) ajoute les balises de fin implicites qui manquaient :

```
<html><head></head><body><ul><li>item </li></ul>
  </body></html>
```

Créer un arbre

```
HTML::TreeBuilder->new_from_file(..)
HTML::TreeBuilder->new_from_content(..)
```

La création de l'arbre se fait à partir du document HTML en utilisant l'une des méthodes suivantes :

- `new_from_file( $file )`
- `new_from_content( @args )`

Ces méthodes sont des raccourcis qui combinent la construction d'un nouvel objet, et l'appel à la méthode `parse_file()` (ou `parse()`, à répétition sur chacun des éléments de `@args`) de `HTML::Parser`.

Ces deux méthodes ne permettent pas de définir des options sur l'analyseur avant l'analyse proprement dite.

Une fois l'analyseur créé avec `new()` (et éventuellement configuré en utilisant les méthodes héritées de `HTML::Parser`), il est également possible d'utiliser directement les méthodes `parse_file()` et `parse()` de `HTML::Parser`.

La structure arborescente issue de l'analyse du HTML est relativement lourde en mémoire, et regorge de références circulaires (chaque nœud enfant pointe vers son parent, et réciproquement). Pour effacer proprement la structure de données, il est donc *impératif* d'utiliser la méthode `delete()` sur l'objet racine.

Rechercher un élément

```
$tree->look_down( . . .)
```

Pour des programmes consacrés à l'analyse d'une page web pour en extraire les données utiles, la seule méthode vraiment indispensable à connaître est `look_down()`, qui cherche à partir de l'invoquant les éléments enfants qui correspondent aux critères de recherche. En contexte de liste, `look_down()` renvoie tous les éléments qui correspondent aux critères, et en contexte scalaire, seulement le premier élément (ou `undef`).

Info

Le module `HTML::Element` dispose d'une API très riche, avec de nombreuses méthodes consacrées au fonctionnement de base (création d'éléments, manipulation des balises et de leurs attributs), à la modification de la structure (ajout, insertion, détachement ou suppression d'éléments, clonage d'objets), au formatage et à l'affichage (déboggage, conversion en HTML (propre), XML, texte), et enfin à la structure elle-même (parcours de l'arbre ; recherche à partir d'un nœud ; obtention d'éléments structurels tels que le parent, les ancêtres, les descendants d'un nœud ; comparaisons d'éléments).

La forme générale d'appel est :

```
$elem->look_down( @criteres );
```

où `$elem` est un objet `HTML::Element`, et les critères contenus dans `@criteres` sont de trois types :

- `(attr_name, attr_value)` : cherche les éléments ayant la valeur souhaitée pour l'attribut indiqué. Il faut noter qu'il est possible d'utiliser des attributs internes, tel `_tag`.
- `(attr_name, qr/. . ./)` : cherche les éléments dont la valeur correspond à l'expression régulière fournie pour l'attribut indiqué.
- un `coderef` : cherche les éléments pour lesquels `$coderef->( $element )` renvoie vrai.

La liste de critères est évaluée dans l'ordre. Il faut savoir que les deux premiers types de critères sont presque toujours plus rapides que l'appel à un coderef, aussi il est intéressant de placer les coderefs en fin de liste.

Le code suivant va rechercher les images dont le texte alternatif indique qu'il s'agit de photos et dont la largeur (l'attribut HTML de la balise `<img>`, pas la largeur réelle du fichier image) est supérieure à 400 pixels.

```
@photos = $elem->look_down(
  _tag => "img",
  alt  => qr/photo/i,
  sub { $_[0]->attr('width') >= 400 },
);
```

`HTML::TreeBuilder` est donc un module de très haut niveau pour analyser et manipuler un document HTML. Les exemples codés avec `HTML::Parser` et `HTML::TokeParser` vont s'en trouver considérablement réduits.

Extraire du texte d'un document avec HTML::TreeBuilder

```
use HTML::TreeBuilder;

my $root = HTML::TreeBuilder->new();
$root->parse($_) while <>;
$root->eof();

print $root->as_trimmed_text;
$root->delete();
```

Difficile de faire plus court...

Produire une table des matières avec HTML::TreeBuilder

```
use HTML::TreeBuilder;

my $root = HTML::TreeBuilder->new();
$root->parse($_) while <>;
$root->eof();

my @elems = $root->look_down(
  _tag => qr/^h[1-6]$/ );
```

```
for my $elem (@elems) {
    my $level = substr($elem->tag, -1, 1);
    print "  " x ( $level - 1 ),
      $elem->as_trimmed_text, "\n";
}

$root->delete();
```

Extraire le titre d'un document avec HTML::TreeBuilder

```
use HTML::TreeBuilder;

my $root = HTML::TreeBuilder->new();
$root->parse($_) while <>;
$root->eof();

print $_->as_trimmed_text, "\n"
  for $root->look_down(_tag => "title");

$root->delete();
```

Astuce

Dans cet exemple, la boucle `for` permet de ne pas traiter le cas où il n'y a pas de balise `<title>` comme un cas particulier. Puisqu'il y a au plus une balise `<title>` dans un document HTML, il est possible d'utiliser `look_down()` en contexte scalaire et d'écrire :

```
print $root->look_down( _tag => 'title' )->
    as_trimmed_text, "\n";
```

En contexte scalaire, `look_down()` va renvoyer `undef` en l'absence de balise `<title>`, ce qui impose de gérer ce cas particulier, sous peine de voir le programme s'interrompre avec le message d'erreur :

```
Can't call method "as_trimmed_text" on an undefined
value.
```

21

HTTP et le Web

HTTP, le protocole de transfert à la base du Web est devenu omniprésent. Parce qu'il est le seul protocole dont il est quasi certain qu'il passera à travers les *proxies* et les *firewalls*, celui-ci est utilisé comme protocole de transport générique pour tous les usages. Non seulement pour le « transport de documents hypertexte » (HTTP signifie *Hyper-Text Transport Protocol*), mais aussi pour encapsuler des échanges entre applications ou des appels de procédures distantes (*web services*, SOAP, XML-RPC, etc.)

La maîtrise du protocole HTTP est donc indispensable pour un programme qui sera amené à faire des échanges avec le monde extérieur.

La bibliothèque *libwww-perl* a été écrite en 1994 pour Perl 4 par Roy Fielding, l'un des principaux auteurs de la spécification HTTP et membre fondateur de la fondation Apache. La version pour Perl 5 (orientée objet) a été écrite par Gisle Aas, et est connue sous le nom de `LWP`. C'est elle (et les modules qui la composent et s'appuient dessus) qui sera décrite dans ce chapitre et le suivant.

Ce chapitre présente ce qui se passe lorsque qu'un navigateur se connecte à un serveur web, et met les différents éléments décrit en relation avec la bibliothèque Perl correspondante.

Adresses

```
http://www.perdu.com:80/index.html
```

Tout commence par une adresse Internet, autrement dit une URL[1] (ou URI[2]). Cette URI peut être découpée en plusieurs éléments :

- `http://` (*scheme*) : le protocole à utiliser (ici HTTP). Les URI servant à décrire des adresses plus générales, il existe des URI associées à d'autres protocoles (FTP, IRC, etc.).
- `www.perdu.com` (*host*) : l'adresse du serveur au sens du DNS.
- `:80` (*port* : le port (TCP) sur lequel écoute le serveur web. Celui est optionnel, et s'il est omis, la valeur par défaut `80` est utilisée.
- `/index.html` (*path*) : le chemin du document sur le serveur.

En Perl, la bibliothèque de gestion des URI s'appelle tout simplement `URI`. Voici un exemple d'utilisation :

```
use URI;

while (<>) {
  chomp;
  my $uri = URI->new($_);
  say join "\n",
    map { "$_:\t" . $uri->$_ }
      qw( scheme host port path );
}
```

1. *Uniform Resource Locator.*
2. *Uniform Resource Identifier.*

Ce code donnera le résultat suivant quand il reçoit l'URL précédente sur l'entrée standard :

```
scheme:  http
host:  www.perdu.com
port:  80
path:  /index.html
```

La classe `URI` fournit de nombreuses méthodes pour manipuler les URI, et dispose de sous-classes adaptées aux divers *schemes*. Ainsi, dans l'exemple précédent, l'objet `$uri` renvoyé par `URI->new()` était de classe `URI::http`.

Messages

Équipé de l'adresse du document à télécharger, le client va se charger de l'obtenir auprès du serveur. Il établit donc une connexion au port indiqué et va dialoguer avec le serveur en utilisant le protocole HTTP. HTTP est un protocole à base de messages. Les messages envoyés par le client sont appelés « requêtes », et les messages envoyés par le serveur sont appelés « réponses ».

Dans les premières versions d'HTTP, la connexion TCP servait à échanger une seule requête et une seule réponse. Ce genre d'inefficacité a été corrigé par la suite (en HTTP /1.1, les connexions TCP entre client et serveur peuvent être persistantes et transporter plusieurs couples requête-réponse). Mais le principe reste le même et il n'y a pas vraiment d'échange : le client envoie l'intégralité de la requête, et ensuite seulement le serveur envoie l'intégralité de la réponse[3].

3. Du point de vue du serveur, deux requêtes successives sur la même connexion TCP peuvent venir de clients différents. HTTP permet en effet l'utilisation de serveurs mandataires (*proxies*), et une requête peut donc transiter de façon automatique à travers un ou plusieurs intermédiaires (parfois à l'insu du client, par exemple dans le cas où un *proxy transparent* intercepte la connexion TCP du client).

Les messages HTTP sont divisés en trois parties :

- **La ligne de statut.** C'est la première ligne du message.

  Pour une requête, elle contient la méthode, ses paramètres, et la version du protocole reconnue par le client.

  Pour une réponse, la ligne de statut contient un code d'erreur, un message en texte clair, et la version du protocole reconnue par le serveur.

- **Les en-têtes.** Les en-têtes se présentent sous la forme d'une liste de clés/valeurs séparées par le caractère `:`. Ce format est similaire à celui des en-têtes d'un courriel.

- **Le corps.** Le corps (optionnel) du message est séparé des en-têtes par une ligne vide.

  Certaines requêtes n'ont pas de corps de message, par définition. C'est le cas des requêtes utilisant les méthodes GET et HEAD, par exemple. D'autres ont un corps de message, comme les requêtes POST.

Dans `LWP`, la classe `HTTP::Message` représente les messages. Les en-têtes peuvent être manipulés *via* la classe `HTTP::Headers`.

Requêtes

```
GET / HTTP/1.1
```

L'observation de ce qui se passe sur le réseau, avec un outil comme Wireshark, montre que le client envoie ceci :

```
GET /index.html HTTP/1.1
Host: www.perdu.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; fr;
  rv:1.9.2.3) Gecko/20100423 Ubuntu/10.04 (lucid)
  Firefox/3.6.3
```

```
Accept: text/html,application/xhtml+xml,application/xml;
  q=0.9,*/*;q=0.8
Accept-Language: fr-fr,fr;q=0.8,en-gb;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset:  ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
```

Il s'agit d'une requête `GET` qui, par définition, n'a pas de corps de message.

La classe `HTTP::Request` de `LWP` hérite de `HTTP::Message`.

Réponses

```
HTTP/1.1 200 OK
```

Voici à quoi ressemble la réponse envoyée par le serveur :

```
HTTP/1.1 200 OK
Date: Tue, 01 Jun 2010 21:19:50 GMT
Server: Apache
Last-Modified: Tue, 02 Mar 2010 18:52:21 GMT
Etag: "4bee144-cc-dd98a340"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 163
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/html

<163 octets de données compressée>
```

Le passage des données compressées contenues dans le corps du message à travers *gunzip* fait apparaître le corps du message, ici le source HTML de la page :

```
<html>
<head>
<title>Vous Etes Perdu ?</title>
</head>
<body>
<h1>Perdu sur l'Internet ?</h1>
<h2>Pas de panique, on va vous aider</h2>
<strong>
<pre>    * <----- vous &ecirc;tes ici</pre>
</strong>
</body>
</html>
```

C'est le navigateur qui se charge du rendu de ce document.

La classe `HTTP::Response` de `LWP` hérite de `HTTP::Message`.

22

LWP

LWP est une bibliothèque très riche (plus de 50 modules), et ce chapitre ne fait que montrer les principaux modules et méthodes disponibles pour réaliser une navigation simple.

Utiliser LWP::Simple

Pour les utilisateurs ayant des besoins très limités, ou désirant écrire des unilignes jetables, la bibliothèque LWP::Simple fournit une vue simplifiée de libwww-perl, en exportant quelques fonctions réalisant des tâches courantes.

Il n'est pas nécessaire de créer un objet URI, LWP::Simple s'en chargera, en appelant URI-new() avec la chaîne de caractères fournie en paramètre.

Faire une requête GET sur une URL

```
get( $url )
```

Comme son nom l'indique, get() fait une requête GET sur l'URL fournie en paramètre, et renvoie le contenu de la réponse associée.

L'URL est une simple chaîne de caractères :

```
use LWP::Simple;
$content = get(shift);
```

La fonction `getprint()` affiche le contenu de la réponse associée à l'URL fournie et renvoie le code HTTP de la réponse,

Enregistrer le contenu de la réponse

```
getstore( $url => $fichier )
```

Cette fonction enregistre le contenu de la réponse associée à l'URL fournie dans le fichier spécifié et renvoie le code HTTP de la réponse.

Faire une requête HEAD sur une URL

```
head( $url )
```

Lorsqu'un programme n'a pas besoin du corps de la réponse, il est inutile (et coûteux au niveau réseau) de le télécharger pour le jeter ensuite. La méthode `HEAD` permet de demander à un serveur web de faire le même traitement qu'il aurait effectuer avec une requête GET, mais de ne renvoyer que les en-têtes de la réponse.

En contexte de liste, la fonction `head()` renvoie quelques valeurs utiles issues de la réponse, ou la liste vide en cas d'échec :

```
my ( $content_type , $document_length ,
  ➥ $modified_time , $expires , $server )
     ➥  = head($url);
```

En contexte scalaire, `head()` renvoie une valeur vraie (en fait, l'objet `HTTP::Response`) en cas de succès.

Cela peut servir par exemple à tester une liste de liens pour filtrer les liens morts :

```
use LWP::Simple;

for my $url (@url) {
  say $url if head($url);
}
```

Utiliser LWP::UserAgent

Dès que le programme nécessite plus de contrôle sur les requêtes et les réponses (par exemple pour manipuler les en-têtes), il est nécessaire de passer par la classe principale de la bibliothèque LWP : `LWP::UserAgent`.

`LWP::UserAgent` implémente un navigateur complet, qui sait manipuler les objets requêtes et réponses présentés page 391 et permet de contrôler dans les moindres détails les requêtes envoyées.

Créer un agent LWP::UserAgent

```
LWP::UserAgent->new( . . .)
```

La méthode `new()` permet de créer un nouvel objet `LWP::UserAgent`. Celle-ci prend une liste de paires clé/valeur en paramètres, pour configurer l'état initial du navigateur.

```
my $ua = LWP::UserAgent->new(%options);
```

Parmi les clés importantes, il faut retenir :

- `agent` : permet de définir la valeur de l'en-tête `User-Agent`.
- `cookie_jar` : définit un objet `HTTP::Cookies` qui sera utilisé pour conserver les *cookies* envoyés par les serveurs web.
- `env_proxy` : demande la définition des proxies à partir des variables d'environnement.
- `keep_alive` : si elle est vraie, cette option crée un cache de connexions (`LWP::ConnCache`), la valeur associée définit la capacité maximale du cache de connexions. Le cache de connexion permet au client d'utiliser le mode *keep-alive* de HTTP 1.1, et d'économiser les connexions TCP.
- `show_progress` : si elle vraie, une barre de progression sera affichée sur le terminal au fur et à mesure du traitement des requêtes.

Ces options peuvent également être définies en appelant les méthodes correspondantes sur un objet `LWP::UserAgent` directement.

Gérer les réponses

`HTTP::Response`

Les méthodes qui envoient des requêtes HTTP à un serveur web vont renvoyer des objets `HTTP::Response` en retour. Le programme va pouvoir utiliser les attributs de cet

objet HTTP::Reponse pour interagir avec l'utilisateur ou le serveur web.

En tant que sous-classe de HTTP::Message, HTTP::Reponse, comme HTTP::Request, dispose en particulier des méthodes suivantes :

- headers() : renvoie l'objet HTTP::Headers qui représente les en-têtes du message.
- content() : renvoie le contenu brut du corps du message.
- decoded_content() : renvoie le contenu *décodé* du corps du message, en utilisant le jeu de caractères défini dans le message.

Les principaux attributs d'un objet HTTP::Response sont :

- code : le code de la réponse (200, 301, etc.) ;
- message : le message textuel associé au code de la réponse ;
- status_line : la ligne de statut de la réponse (c'est-à-dire la chaîne "<code> <message>").

Faire une requête GET sur une URL avec LWP::UserAgent

```
$ua->get( $url );
```

Pour les requêtes HTTP qui n'ont pas de corps de message (GET et HEAD, par exemple), les méthodes de LWP::UserAgent peuvent se contenter d'une URI, et ajouteront les en-têtes par défaut. Ici aussi pour l'URI, une chaîne de caractères suffit ; la bibliothèque créera l'objet URI nécessaire de façon transparente.

Cette méthode effectue donc une requête GET à partir de l'URI fournie. La valeur renvoyée est un objet `HTTP::Response`.

```
use LWP::UserAgent;

my $response
    ➥ = LWP::UserAgent->new()->get($url);

print $response->is_success
    ? $response->decoded_content
    : $response->status_line;
```

Un objet `HTTP::Response` est renvoyé dans tous les cas, y compris quand LWP ne peut se connecter au serveur. (Une réponse d'erreur générée par LWP aura la valeur `Internal response` pour l'en-tête `Client-Warning`.)

Il est possible d'ajouter des en-têtes aux en-têtes par défaut, en faisant suivre l'URL d'une liste de paires clé/valeur.

```
my $response = $ua->get( $url,
  ➥ User_Agent => 'Mozilla/3.14159' );
```

Enregistrer le contenu de la réponse

```
:content_file
```

Lors de l'appel à `get()`, les clés qui commencent par `:` sont spéciales et ne seront pas interprétées comme des entêtes à ajouter à la requête. Parmi les clés spéciales reconnues, `:content_file` permet d'indiquer un nom de fichier dans lequel sauver le corps de la réponse. Dans le cas où la réponse n'est pas un succès (code 2xx), le corps de la

réponse sera toujours sauvé dans l'objet `HTTP::Response`. L'équivalent de :

```
use LWP::Simple;

getstore( $url, $filename );
```

sera donc :

```
use LWP::UserAgent;

my $ua = LWP::UserAgent->new();
$ua->get( $url,
  ➥ ':content_file' => $filename );
```

Faire une requête HEAD sur une URL avec LWP::UserAgent

```
$ua->head( $url );
```

Exactement comme `get()`, `head()` va créer une requête HEAD à partir de l'URL fournie, et va renvoyer l'objet `HTTP::Response` correspondant.

Un programme de test de lien pourra fournir un peu plus d'informations sur l'état des liens (ici, le code HTTP de la réponse) :

```
use LWP::UserAgent;

my $ua = LWP::UserAgent->new();

for my $url (@url) {
  my $response = $ua->head($url);
  say $reponse->code, ' ', $url;
}
```

Faire une requête POST sur une URL avec LWP::UserAgent

```
$ua->post( $url, . . .);
```

Les requêtes POST sont des messages avec un corps. Elles servent typiquement lors de la validation de formulaires HTML, et lors de l'envoi de fichiers.

Cette méthode prend toujours l'URL comme premier paramètre. Celle-ci et les paramètres qui suivent sont passés à la fonction `POST HTTP::Request::Common`, à laquelle est déléguée la création de l'objet `HTTP::Request` représentant la requête POST.

En plus des paires clé/valeur permettant de définir des entêtes spécifiques, elle accepte une référence de tableau ou de hash pour définir le contenu d'un formulaire web.

Le code suivant :

```
$ua->post(
  'http://www.example.com/submit.cgi',
  [ couleur => 'rouge',
    fruit   => 'banane',
    nombre  => 3
  ]
);
```

produira la requête suivante :

```
POST http://www.example.com/submit.cgi
Content-Length: 35
Content-Type: application/x-www-form-urlencoded

couleur=rouge&fruit=banane&nombre=3
```

Le résultat sera identique avec une référence de hash.

Envoyer des requêtes

```
$ua->request( $request );
```

Les méthodes `get()`, `head()` et `post()` décrites précédemment sont en fait des « raccourcis » pour la méthode générale d'envoi de requêtes `request()`. Celle-ci prend en premier paramètre un objet `HTTP::Request`, que les raccourcis créaient à partir des paramètres fournis.

Une autre méthode générale existe, `simple_request()`. À la différence de `request()`, elle ne suit pas automatiquement les redirections, et sera donc susceptible de renvoyer un objet `HTTP::Response` avec un code 3xx.

Différences entre LWP::UserAgent et un « vrai » navigateur

L'URL http://www.perdu.com/ a déjà servi d'exemple, page 389. Le code ci-dessous affiche le texte de la requête envoyée par `LWP::UserAgent` et celui de la réponse reçue.

```
use LWP::UserAgent;

my $ua  = LWP::UserAgent->new();
my $uri = URI->new(
  shift // "http://www.perdu.com/"
);

# la requête
my $req = HTTP::Request->new(GET => $uri);
say $req->as_string;

# la réponse
my $res = $ua->request($req);
say $res->as_string;
```

Ce code affiche le résultat suivant :

```
GET http://www.perdu.com/index.html

HTTP/1.1 200 OK
Connection: close
Date: Fri, 04 Jun 2010 22:56:04 GMT
Accept-Ranges: bytes
ETag: "4bee144-cc-dd98a340"
Server: Apache
Vary: Accept-Encoding
Content-Length: 204
Content-Type: text/html
Last-Modified: Tue, 02 Mar 2010 18:52:21 GMT
Client-Date: Fri, 04 Jun 2010 22:56:04 GMT
Client-Peer: 208.97.189.107:80
Client-Response-Num: 1
Title: Vous Etes Perdu ?
X-Pad: avoid browser bug

<html><head><title>Vous Etes Perdu ?</title></head>
<body><h1>Perdu sur l'Internet ?</h1><h2>
Pas de panique, on va vous aider</h2><strong><pre>
    * <----- vous &ecirc;tes ici</pre></strong>
</body></html>
```

La première différence notable avec la réponse qui a été envoyée à un navigateur « normal » (voir page 393), c'est que le corps de la réponse n'est pas compressé. Mais il y a d'autres différences...

La capture du trafic réseau permet de voir la requête que `LWP::UserAgent` a effectivement envoyé :

```
GET /index.html HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: www.perdu.com
User-Agent: libwww-perl/5.834
```

Celle-ci est remarquablement plus courte que celle de Firefox (voir page 392). Les différences entre les deux requêtes expliquent les différences entre les réponses. Par exemple, en envoyant l'en-tête `Accept-Encoding`, Firefox signale qu'il accepte un certain nombre d'encodages pour la réponse. La réponse du serveur contenait l'en-tête `Content-Encoding: gzip` qui indiquait que le corps (*et le corps seulement*) de celle-ci était encodé avec *gzip*. L'utilisation de la compression permet d'économiser de la bande passante, quand client et serveur peuvent s'entendre sur l'encodage.

Tant qu'à scruter le réseau, voici également les en-têtes de la réponse telle qu'elle a circulé :

```
HTTP/1.1 200 OK
Date: Fri, 04 Jun 2010 22:56:04 GMT
Server: Apache
Last-Modified: Tue, 02 Mar 2010 18:52:21 GMT
ETag: "4bee144-cc-dd98a340"
Accept-Ranges: bytes
Content-Length: 204
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug
```

Les en-têtes suivants n'ont pas été envoyés par le serveur !

```
Client-Date: Fri, 04 Jun 2010 22:56:04 GMT
Client-Peer: 208.97.189.107:80
Client-Response-Num: 1
Title: Vous Etes Perdu ?
```

Ces en-têtes ont été ajoutés par LWP : les en-têtes `Client-` fournissent quelques informations supplémentaires sur l'échange, et l'en-tête `Title` a été obtenu en lisant le bloc `<head>` du contenu de la réponse HTML.

Un point important à retenir de cet exemple, et à garder à l'esprit lors du déboggage d'un programme, c'est qu'une URI ne suffit pas pour connaître la réponse exacte qui sera envoyée. Celle-ci dépend également des en-têtes de la requête. Et un navigateur courant peut envoyer *beaucoup* d'en-têtes, qui seront interprétés par le serveur et influeront sur le contenu de la réponse.

23

Navigation complexe

Écrire un robot pour naviguer sur le Web n'a pas pour objectif de récupérer une simple page. Il s'agit en général de rejouer une navigation plus ou moins complexe, pour mettre à jour des données ou télécharger des fichiers.

La complexité peut prendre plusieurs formes, dont les plus courantes sont décrites dans ce chapitre.

Traiter les erreurs

```
$response->is_success()
```

Les réponses HTTP sont séparées en cinq différentes classes. Le premier des trois chiffres du code de la réponse indique sa classe. `HTTP::Response` dispose de méthodes booléennes qui permettent de savoir si une réponse appartient à une classe donnée. Les différentes classes sont :

- `1xx` : Information : requête reçue, le traitement continue.

```
$response->is_info
```

- `2xx` : Succès : l'action requise a été reçue, comprise et acceptée avec succès.

```
$response->is_success
```

- `3xx` : Redirection : une action supplémentaire doit être effectuée pour compléter la requête.

```
$response->is_redirect
```

- `4xx` : Erreur du client : la requête a une syntaxe incorrecte ou ne peut être effectuée.

```
$response->is_error
```

- `5xx` : Erreur du serveur : le serveur a été incapable de répondre à une requête apparemment correcte.

```
$response->is_error
```

  (La méthode est la même pour les erreurs client et les erreurs serveur.)

Cela permet de tester de façon générique le résultat d'une requête, sans regarder le détail de la réponse.

Authentifier

```
$ua->credentials( . . .)
```

L'accès à certains documents peut être limité à un petit nombre d'utilisateurs, et protégé par le mécanisme d'authentification inclus dans le protocole HTTP.

Lorsqu'une ressource est protégée par mot de passe, le serveur renvoie une réponse d'erreur (`401 Authentication required`). Il est possible de laisser `LWP::UserAgent` gérer

l'authentification directement, en l'informant des codes d'authentification liés à un site et aux domaines associés (*realm* en anglais).

Les informations d'authentifications (nom d'utilisateur et mot de passe) sont associées à l'emplacement réseau (la chaîne `"host:port"`) et au domaine d'authentification (le *realm* dans le jargon HTTP).

Le programme suivant teste la fonction sur le site **http://diveintomark.org/tests/client/http/** qui contient toute une batterie de tests pour les clients HTTP :

```
use LWP;
my $url = 'http://diveintomark.org/tests/'
        . 'client/http/200_basic_auth.xml';
my $ua = LWP::UserAgent->new();

$ua->credentials(
  'diveintomark.org:80' => 'Use test/basic',
  'test' => 'basic'
);

my $response = $ua->get($url);
say $response->status_line;
```

Si l'appel à `credentials()` est commenté, le programme affiche :

```
401 Authorization Required
```

sinon, il affiche :

```
200 OK
```

`LWP::UserAgent` supporte les méthodes d'authentification *Basic*, *Digest* et *NTLM*.

Gérer les cookies

HTTP::Cookies

Les *cookies* sont un moyen d'associer un état (en fait une chaîne de caractères, qui sera décodée par le serveur web) à un site et un chemin dans le site. Certains sites utilisent des *cookies* pour maintenir une session utilisateur, aussi une navigation enchaînant plusieurs requêtes peut donc nécessiter de maintenir les *cookies* entre les requêtes, afin que le serveur soit en mesure d'associer celles-ci au même « utilisateur ».

Normalement, le serveur web crée le cookie en envoyant un en-tête `Set-Cookie:` ou `Set-Cookie2:` avec sa réponse. Le client envoie les cookies qu'il possède avec l'en-tête `Cookie:`.

La classe `HTTP::Cookies` sert à manipuler un *cookie jar* (une « boîte à gâteaux »), qui sera utilisé par l'objet `LWP::UserAgent` pour analyser les en-têtes de réponse modifiant les cookies, et mettre à jour les en-têtes associés dans les requêtes.

Il est possible d'initialiser le navigateur avec un *cookie jar* vide, comme ceci :

```
my $ua = LWP::UserAgent->new(
  cookie_jar => {}
);
```

`HTTP::Cookies` permet de sauvegarder automatiquement puis de relire les cookies depuis un fichier :

```
use HTTP::Cookies;

my $cookie_jar = HTTP::Cookies->new(
  file     => "$ENV{HOME}/cookies.dat",
  autosave => 1,
);
```

Il est ensuite possible d'utiliser ce *cookie jar* :

```
$ua->cookie_jar($cookie_jar);
```

Grâce au paramètre `autosave`, les cookies seront automatiquement enregistrés lors de la destruction de l'instance de `HTTP::Cookies`.

Créer un objet HTML::Form

```
HTML::Form->parse( . . .)
```

La méthode de classe `parse()` analyse un document HTML et construit des objets `HTML::Form` pour chacun des éléments `<form>` trouvés dans le document. En contexte scalaire, elle renvoie le premier formulaire.

Formulaires

L'un des intérêts de l'utilisation de LWP est l'automatisation de tâches fastidieuses nécessitant un navigateur web. Il devient possible d'ajouter une interface scriptable au-dessus d'un banal site web.

Nombre de tâches consistent à remplir un formulaire avec des données diverses. Si ces données peuvent être obtenues par un programme, il est particulièrement utile de permettre à ce programme de remplir le formulaire lui-même.

Le module `HTML::Form` permet de produire des objets de type formulaire à partir de source HTML, et de remplir ces formulaires. Leur validation renverra un objet requête qui pourra être utilisé afin d'obtenir une réponse.

La méthode `dump()` renvoie une version texte de l'état du formulaire, ce qui peut être utile lors du développement d'un robot.

Le script suivant affiche le formulaire contenu dans une page web fournie en paramètre.

```
use LWP;
use HTML::Form;

say my $url = shift;
my $response
   ➥= LWP::UserAgent->new()->get($url);
die $response->status_line
    if !$response->is_success;

say $_->dump
  for HTML::Form->parse($response,
   ➥base => $url);
```

Exécuté sur la page d'accueil d'un moteur de recherche bien connu, celui-ci affiche :

```
GET http://www.google.com/search [f]
  hl=fr                    (hidden readonly)
  source=hp                (hidden readonly)
  ie=ISO-8859-1            (hidden readonly)
  q=                       (text)
  btnG=Recherche Google    (submit)
  btnI=J'ai de la chance   (submit)
```

Sélectionner et modifier des champs de formulaire

```
$form->value( $selector, $value )
```

Un objet `HTML::Form` est une suite de champs (objets de la classe `HTML::Form::Input`).

Les champs sont obtenus à l'aide de sélecteurs. Un sélecteur préfixé par `#` doit correspondre à l'attribut `id` du champ. S'il est préfixé par `.` il doit correspondre à son attribut `class`. Le préfixe `^` (ou l'absence de préfixe) correspond à une recherche sur le nom du champ.

Cette méthode est en fait un raccourci, qui combine un appel à la méthode `find_input()` de `HTML::Form` (qui renvoie un objet `HTML::Form::Input`) et à la méthode `value()` de `HTML::Form::Input`.

Les deux lignes de code suivantes sont exactement équivalentes :

```
$form->value($selector, $value);

$form->find_input($selector)->value($value);
```

Valider un formulaire

```
$form->click()
```

Renvoie l'objet `HTTP::Request` correspondant à la validation du formulaire. Celui-ci peut ensuite être utilisé avec la méthode `request()` de `LWP::UserAgent`.

24

WWW::Mechanize

L'utilisation de `LWP::UserAgent` (voir page 397) peut se révéler fastidieuse pour l'écriture de robots devant réaliser des navigations longues et complexes, nécessitant contrôle d'erreur, authentification, utilisation de cookies et validation de formulaires. Ces opérations exigent l'ajout de beaucoup de code structurel, qui est nécessaire mais répétitif, et obscurcit le déroulement du programme.

`WWW::Mechanize` est une sous-classe de `LWP::UserAgent` (ce qui permet le cas échéant de revenir à la classe de base si besoin est), qui fournit des méthodes adaptées à l'écriture de robots. Le code structurel est caché dans des méthodes de haut niveau, rendant le code source des robots plus court et plus lisible, puisqu'il ne contient que les opérations qui font sens au niveau de la navigation.

Créer un objet `WWW::Mechanize`

```
WWW::Mechanize->new( %options )
```

Le constructeur de `WWW::Mechanize` supporte les options du constructeur de `LWP::UserAgent`, et en ajoute quelques-unes.

Parmi elles, l'option `autocheck` (activée par défaut) va forcer `WWW::Mechanize` à vérifier que chaque requête s'est déroulée correctement (c'est-à-dire que la réponse a un code en `2xx`). En général, il n'y a donc pas besoin de vérifier que les requêtes ont débouché sur un succès.

```
$ perl -MWWW::Mechanize \
-e 'WWW::Mechanize->new->get(shift)' \
http://www.example.com/rien

Error GETing http://www.example.com/rien:
    Not Found at -e line 1
```

Dans le cas où le robot doit traiter les erreurs de manière plus fine, il faudra désactiver `autocheck` et faire la validation manuellement.

L'option `stack_depth` permet de définir la taille de l'historique du navigateur. Par défaut, celle-ci n'est pas limitée, mais un robot faisant beaucoup de requêtes risque de consommer beaucoup de mémoire. Si `stack_depth` est initialisée à `0`, aucun historique n'est conservé.

`WWW::Mechanize` active également la gestion des cookies automatiquement.

L'exemple ci-dessous change les valeurs par défaut :

```
use WWW::Mechanize;

my $m = WWW::Mechanize->new(
  # pas de vérification automatique
  autocheck => 0,

  # mémoire de 5 pages
  stack_depth => 5,

  # pas de cookies
  cookie_jar => undef,
);
```

Lancer une requête GET

```
$m->get( $url )
```

Cette méthode fait un GET sur l'URL en paramètre et renvoie un objet `HTTP::Response`.

L'objet `HTTP::Response` correspondant à la dernière requête est également accessible *via* la méthode `response()`.

Lancer une requête POST

```
$m->post( $url, . . .)
```

`WWW::Mechanize` étant une sous-classe de `LWP::UserAgent`, il possède également une méthode `post()` (voir page 402 pour les détails).

Revenir en arrière

```
$m->back()
```

Cette méthode est l'équivalent de l'appui sur le bouton Back d'un navigateur. Renvoie une valeur booléenne indiquant le succès ou l'échec de l'opération.

Elle ne rejoue pas la requête : rien n'est envoyé sur le réseau. Elle se contente de remettre le robot dans l'état où il se trouvait avant la dernière requête.

Recharger une page

```
$m->reload()
```

Cette méthode rejoue la dernière requête (nouvel envoi d'une requête identique à la précédente) au serveur. N'altère pas l'historique.

Suivre des liens

```
$m->follow_link( . . .)
```

Avec les méthodes décrites ci-après, la puissance d'expression de `WWW::Mechanize` commence à apparaître.

`find_link()` permet de trouver un lien dans la page en cours. Les critères de recherche permettent de tester l'égalité ou la correspondance avec une expression régulière.

Les critères sont :

- `text`, `text_regex` : le texte du lien.
- `url`, `url_regex` : l'URL du lien (telle que trouvée dans le document).
- `url_abs`, `url_abs_regex` : l'URL absolue du lien (les URL relatives étant automatiquement converties).
- `name`, `name_regex` : le nom du lien.
- `id`, `id_regex` : l'attribut `id` du lien.
- `class`, `class_regex` : l'attribut `class` du lien.
- `tag`, `tag_regex` : la balise du lien (la balise `<a>` n'est pas la seule à contenir des liens).
- n : le nième lien correspondant.

Si plusieurs critères sont fournis, le lien renvoyé devra les respecter tous.

Quelques exemples :

- `text_regex => qr/download/` : le texte lien doit correspondre à l'expression régulière `qr/download/`.
- `text => bananana, url_regex => qr/spelling/` : le texte du lien doit être `"bananana"`, et l'URL associée doit correspondre à l'expression régulière `qr/spelling/`.
- `text => 'Cliquez ici', n => 4` : le lien doit être le quatrième lien dont le texte est `"Cliquez ici"`.

Par défaut `n` est à `1`, c'est donc le premier lien correspondant qui sera renvoyé. Le cas particulier `n => 'all'` demande de renvoyer l'ensemble des liens correspondants.

L'objet renvoyé est de la classe `WWW::Mechanize::Link`. La partie intéressante est son `url()`.

```
use WWW::Mechanize;
my $m = WWW::Mechanize->new();
$m->get( $url );

# trouve le premier lien qui parle de banane
my $link = $m->find_link(
  ➥ text_regex => qr/banane/ );

# et affiche son URL
say $link->url;
```

`follow_link()` est un raccourci pour le cas le plus courant : elle fait simplement un `get()` sur le résultat de `find_link()`.

Info

`WWW::Mechanize` permet de simplifier énormément la gestion de la complexité décrite page 407. Comme le montrent les sections suivantes, traitement d'erreur, gestion des cookies et des formulaires sont facilités.

Traiter les erreurs

Comme on l'a vu, `WWW::Mechanize` fait par défaut une gestion d'erreur minimale : le programme meurt quand la réponse reçue par `get()` n'est pas un succès.

La méthode `success()` permet de tester directement le succès d'une réponse, si le navigateur a été créé sans l'option `autocheck`.

```
my $m = WWW::Mechanize->new(autocheck => 0);
$m->get($url);

if ( !$m->success ) {
    # traitement de l'erreur
}
```

Authentifier

```
$m->credentials( $username, $password )
```

La méthode `credentials` définit le nom d'utilisateur et le mot de passe à utiliser jusqu'à nouvel ordre.

Gérer les cookies

Les cookies sont gérés automatiquement par `WWW::Mechanize`.

Il est toujours possible d'utiliser le paramètre `cookie_jar` du constructeur pour forcer par exemple l'utilisation de cookies sauvegardés dans un fichier (voir page 410).

Gérer les formulaires

```
$m->forms
```

La méthode `forms()` renvoie la liste des objets `HTML::Form` associés aux formulaires de la page en cours.

Lorsqu'il faut afficher des formulaires intermédiaires au cours de la navigation (typiquement lors de l'écriture d'un programme complexe), il est utile d'employer la formulation suivante :

```
print $_->dump() for $m->forms();
```

La plupart des méthodes de `WWW::Mechanize` associées aux formulaires travaillent sur le formulaire courant, qui a été sélectionné au moyen d'une des méthodes décrites ci-après. Si aucun formulaire n'a été sélectionné, le formulaire par défaut est le premier de la page.

Sélectionner un formulaire par son rang

```
$m->form_number($number)
```

Sélectionne le nième formulaire de la page. Attention, la numérotation commence à `1`.

Sélectionner un formulaire par son nom

```
$m->form_name( $name )
```

Sélectionne le premier formulaire portant le nom donné.

Sélectionner un formulaire par son identifiant

```
$m->form_id( $id )
```

Sélectionne le premier formulaire ayant l'attribut `id` donné.

Sélectionner un formulaire par ses champs

```
$m->form_with_fields( @fields )
```

Sélectionne le premier formulaire comportant les champs dont les noms sont donnés.

Remplir le formulaire sélectionné

```
$m->set_fields( . . .)
```

Cette méthode modifie plusieurs champs du formulaire courant. Elle prend une liste de paires clé/valeur, représentant les noms des champs et les valeurs à leur appliquer.

```
$m->set_fields(
   username => 'anonyme',
   passowrd => 's3kr3t',
);
```

Valider le formulaire sélectionné

```
$m->submit()
$m->click( $button )
```

Ces deux méthodes valident et envoient le formulaire vers le serveur.

`submit()` envoie le formulaire, sans cliquer sur aucun bouton.

`click()` envoie le formulaire en simulant le clic sur un bouton. Si aucun nom de bouton n'est donné en paramètre, elle simule un clic sur le premier bouton.

Sélectionner, remplir et valider un formulaire

```
$m->submit_form( . . .)
```

Cette méthode combine la sélection du formulaire, son remplissage et sa validation en un seul appel.

Le code suivant :

```
$m->submit_form(
  form_name => 'search',
  fields    => { query => 'Perl' },
  button    => 'Search Now',
);
```

sera équivalent à :

```
$m->form_name( 'search' );
$m->fields( query => 'Perl' );
$m->click( 'Search Now' );
```

Exemple d'application

Pour illustrer l'utilisation de `WWW::Mechanize`, l'exemple d'un site de « *paste* » va être utilisé. Ces sites web permettent de coller du texte dans un formulaire simple, et renvoient un URL qu'il suffit ensuite de faire suivre à un correspondant pour qu'il puisse visualiser le texte en question. Ce genre de site est très utilisé sur IRC (en particulier sur les canaux consacrés à la programmation), afin d'éviter de polluer les canaux avec des lignes et des lignes de code. Certains sites sont connectés à un robot qui annonce les nouveaux ajouts sur le canal sélectionné.

C'est le site **http://nopaste.snit.ch** qui sera utilisé[1].

Le programme suivant se connecte au site et y sauvegarde le contenu de l'option *–paste* ou à défaut le contenu de l'entrée standard, ce qui permet de l'utiliser à la fin d'une ligne de commande Unix.

```
use strict;
use warnings;
use 5.010;
use WWW::Mechanize;
use Getopt::Long;

my $paste = 'http://nopaste.snit.ch/';

# récupération des options
my %option = (
  channel => '',
  nick    => '',
  summary => '',
  paste   => '',
  list    => '',
);
```

1. Il en existe de nombreux autres, utilisant la même interface ou une interface similaire. Le module CPAN unifiant l'accès aux sites de *paste* est `App::Nopaste`.

```
GetOptions(
  \%option,     'nick=s',
  'summary=s', 'paste=s',
  'channel=s', 'list!',
) or die "Mauvaises options";

# création du robot et connexion au site
my $m = WWW::Mechanize->new;
$m->get($paste);

# affiche la liste des canaux
if ( $option{list} ) {
  print "Canaux disponibles :\n",
      map "- $_\n", grep $_,
      $m->current_form()
      ->find_input('channel')
      ->possible_values;
  exit;
}

# sans option paste,
# lit les données sur l'entrée standard
if ( !$option{paste} ) {
  $option{summary} ||= $ARGV[0] || '-';
  $option{paste} = join "", <>;
}

# remplit et valide le formulaire
# note: list n'est dans le formulaire
delete $option{list};
$m->set_fields(%option);
$m->click;

# affiche l'URL donnée en réponse
print +( $m->links )[0]->url, "\n";
```

L'option `list`, quand elle est activée, fournit la liste des canaux IRC disponibles en affichant les valeurs possibles du champ associé du formulaire (obtenues à l'aide des méthodes de `HTML::Form`).

JavaScript

De nombreux sites utilisent JavaScript pour faire exécuter du code par le client, le plus souvent lié à la présentation, mais pas seulement.

Ainsi, le code JavaScript peut créer des cookies (normalement, les cookies sont envoyés par le serveur) qui ne seront pas détectables par `LWP::UserAgent`. La technologie dite « AJAX » va ainsi créer des requêtes supplémentaires pour obtenir des données (en général au format JSON) qui seront utilisées par le code JavaScript pour modifier les données affichées.

En général, il n'y a pas besoin d'exécuter le JavaScript pour simuler la navigation d'un utilisateur humain. La lecture du code JavaScript et l'analyse du trafic HTTP suffisent souvent à découvrir quels requêtes ou en-têtes supplémentaires sont nécessaires.

Néanmoins, certains sites (trop modernes ou mal conçus) ne sont véritablement fonctionnels qu'avec un navigateur disposant d'un support JavaScript. Il existe toutefois plusieurs solutions, disponibles sur le CPAN, pour naviguer mécaniquement avec exécution du code JavaScript.

Les utilisateurs du système Windows peuvent déjà essayer avec `Win32::IE::Mechanize`, qui permet de piloter Internet Explorer au travers du protocole OLE. Mais ce module n'est toutefois plus vraiment maintenu.

Les utilisateurs de systèmes Unix peuvent eux regarder les modules `Mozilla::Mechanize` et `Gtk2::WebKit::Mechanize`, qui tous les deux s'appuient sur le toolkit Gtk2 et son support pour embarquer un moteur de navigation, dans le premier cas Gecko, le moteur de Mozilla Firefox, et dans le second cas

WebKit, le moteur de Safari et Chrome. Le défaut de ces modules est qu'ils sont écrits en XS et nécessitent donc de disposer des sources des moteurs en question, ce qui les rend difficiles à installer.

`WWW::Mechanize::Firefox` est de ce point de vue déjà plus facile à mettre en place car il s'appuie sur le plugin MozRepl de Firefox qui permet de contrôler le navigateur à distance par un simple protocole. Le module est donc pur Perl, et Firefox étant disponible sur de nombreux systèmes, l'installation et l'utilisation de cette solution est bien plus envisageable.

Il existe encore une dernière solution, `WWW::Scripter`, qui fait partie de ces modules totalement fous (mais dans le bon sens du terme) qui peuplent le CPAN. Il s'agit d'une sous-classe de `WWW::Mechanize` qui utilise des modules implémentant un moteur JavaScript (JE) avec les supports DOM pour HTML et CSS associés. Tous ces modules étant en pur Perl, leur installation ne pose normalement pas de problème particulier. `WWW::Scripter` peut aussi utiliser SpiderMonkey, le moteur JavaScript de Mozilla Firefox, mais le module Perl correspondant a le problème usuelle de la dépendance aux sources du logiciel pour son installation. Il n'est pas impossible qu'il supporte aussi V8, le moteur JavaScript de Chrome, étant donné que `JavaScript::V8` fournit déjà l'interface.

A

Tableau récapitulatif des opérateurs

Cette annexe fournit la liste des opérateurs de Perl, présentés dans l'ordre de précédence.

Précédence	Nom	Description
1	->	Déréférencement, appel de méthode
2	++	Incrémentation de variable
2	-	Décrémentation de variable
3	**	Puissance
4	!	*NON* logique
4	~	*NON* binaire
4	\	Création de référence
5	=~	Application d'une expression régulière
5	!~	Application négative d'une expression régulière
6	*	Multiplication
6	/	Division
6	%	Modulo

Précédence	Nom	Description
6	`x`	Duplication de chaînes
7	`+ - .`	Opérations arithmétiques et concaténation
8	`<<`	Décalage binaire gauche
8	`>>`	Décalage binaire droit
9	`<` et `<=`	Infériorité numérique
9	`>` et `>=`	Supériorité numérique
9	`lt` et `le`	Infériorité de chaîne
9	`gt` et `ge`	Supériorité de chaîne
10	`==`	Égalité numérique
10	`!=`	Non-égalité numérique
10	`<=>`	Comparaison numérique
10	`eq`	Égalité de chaîne
10	`ne`	Non-égalité de chaîne
10	`cmp`	Comparaison de chaînes
11	`&`	*ET* binaire
12	`|`	*OU* binaire
12	`^`	*OU* exclusif binaire
13	`&&`	*ET* logique
14	`||`	*OU* logique
15	`..` et `...`	Création d'intervalles
16	`? :`	Test alternatif
17	`=`	Affectation de variable
17	`+= -= *=` etc.	Modification et affectation
18	`,`	Création de liste
18	`=>`	Création de liste
19	`not`	*NON* logique (faible précédence)
20	`and`	*ET* logique (faible précédence)
21	`or`	*OU* logique (faible précédence)
21	`xor`	*OU* exclusif logique

Index

Symboles

| (regexp) 121
* (opérateur) 24
** (opérateur) 24
+ (opérateur) 24
++ (opérateur) 24
+= (opérateur) 24
- (opérateur) 24
- (opérateur) 24
-= (opérateur) 24
. (opérateur) 25
. (regexp) 100
.. (opérateur) 48, 61
/ (opérateur) 24
$/ (variable prédéfinie) 185–187
=~ (regexp) 97, 98
== (opérateur) 30
? (regexp) 112
?+ (regexp) 115
?? (regexp) 114
$ (regexp) 100, 107, 108
% (opérateur) 24, 89
$_ (variable prédéfinie) 36, 86, 87, 89, 97, 98, 124
^ (opérateur) 73
^ (regexp) 100, 107
\D (regexp) 105
\G (regexp) 109
\K (regexp) 109
\S (regexp) 105
\W (regexp) 105, 108
\Z (regexp) 108
\b (regexp) 108
\d (regexp) 105, 111
\s (regexp) 105, 111
\w (regexp) 105, 108, 111
\z (regexp) 108
> (opérateur) 30
>= (opérateur) 30
< (opérateur) 30
<= (opérateur) 30
-> (opérateur) 40
=> (opérateur) 74
$! (variable prédéfinie) 184, 195
{n,}+ (regexp) 115
{n,}? (regexp) 114
{n,m} (regexp) 113
{n,m}+ (regexp) 115
{n,m}? (regexp) 114
{n} (regexp) 113
{n}+ (regexp) 115
{n}? (regexp) 114
\1 (variable prédéfinie) 117
$1 (variable prédéfinie) 97, 98, 117
\2 (variable prédéfinie) 117
$2 (variable prédéfinie) 97, 98, 117
\3 (variable prédéfinie) 117
$3 (variable prédéfinie) 97, 98, 117

A

abstraction SQL 223
accès à un objet Moose 140
alternative d'expression régulière 121
analyse de document HTML 363
`HTML::Parser`, 365
arrêt, 369
événement, 366
instanciation, 365
lancement, 368
`HTML::TokeParser`
balise, 379
texte, 380, 381
token, 378
`HTML::TreeBuilder`
création, 384
recherche, 385
`HTML::TreeBuilder`, 383
`HTML::TokeParser`, 377
regexp, 364
ancre d'expression régulière
correspondance globale, 109
début de chaîne, 108
début de ligne, 106
fin de chaîne, 108
fin de ligne, 107
frontière de mot, 108
par préfixe, 109
année 259
`AnyEvent::CouchDB` (module) 244
aplatissement
de liste, 61
de tableau, 61
`App::cpanminus` (module) 11
arbre DOM 276
arrêter un programme 42
atome d'expression réguliere 102
attribut
d'élément XML, 293
Moose, 138
avec déclencheur, 150
obligatoire, 144
paresseux, 148
augmentation de méthode parente Moose 177
awk 124

B

base de données 203
abstraction SQL, 223
DBI, 203
connexion, 205
déconnexion, 209
erreur, 216
profiling, 219
test connexion, 208
trace, 217
KiokuDB, 239
administration, 242
récupération, 240
stockage, 240
non-SQL
clé-valeur, 243
document, 244
ORM, 232
BerkeleyDB 239
`bless` (fonction) 132
boucle 35
`foreach`, 35
sur hash, 82, 83
sur liste, 65
sur tableau, 65
`while`, 36

C

chaînage d'attribut Moose 153
chaîne de caractères 20
création, 20

champ de formulaire HTML 412
changement de répertoire 201
chargement de document XML 275, 286
chemin de fichier 189, 190
`chomp` (fonction) 25
`chop` (fonction) 25, 37
`Class::DBI` (module) 232, 233
classe
  d'un objet, 132
  de caractères, 104
  Moose, 137
clé de hash 81
commande système 42
communication entre sessions POE 336
`Config::IniFiles` (module) 320, 322
consommation de rôle Moose 157
constructeur
  d'objet, 135
  Moose, 159
construction
  d'attribut Moose, 147
  d'objet Moose, 159
contenu d'élément XML 292
contexte 22
  de liste, 23
  scalaire de chaine, 23
  scalaire numérique, 22
cookie HTTP 410, 420
copier-coller XML 297
`cos` (fonction) 24
CouchDB 240, 244
`CouchDB::Client` (module) 244
CPAN (Comprehensive Perl Archive Network) 7
cpanm 11
création
  d'objet, 131
  de répertoire, 193
  de session POE, 331
`Cwd` (module) 201

D

déboggage 14
`Data::Dumper` (module) 71-73, 82, 301, 303, 305-307, 316, 361
`Data::Phrasebook::SQL` (module) 226
`Data::Phrasebook` (module) 223, 224, 226
`Data::Phrasebook::SQL` (module) 223-225, 227, 230
date
  du jour, 259
  interprétation, 249
    langue, 251
    str2time, 250
    strptime, 250
date et heure 247
  DateTime
    année, 259
    date du jour, 259
    durée, 263
    formateur, 268
    heure, 261
    interprétation, 270
    intervalle de temps, 263
    jour, 260
    maintenant, 258
    mois, 260
    seconde, 261
  durée, 252, 263
    maintenant, 254
  interprétation de durée, 253
  intervalle de temps, 252, 263
    maintenant, 254
`Date::Language` (module) 251
`Date::Parse` (module) 250, 251
DateTime
  année, 259

DateTime (suite)
calcul
ajout de durée, 265
futur, 264
intervalle, 267
passé, 265
soustraction de durée, 266
date du jour, 259
formateur, 268
heure, 261
interprétation, 270
jour, 260
maintenant, 258
mois, 260
seconde, 261
DateTime (module) 251, 256, 259, 263–265, 267–270
DateTime::Duration (module) 256, 263, 265–267
DateTime::Event (module) 256
DateTime::Format::Oracle (module) 271
DateTime::Format (module) 256, 268
DateTime::Format::Builder (module) 271
DateTime::Format::DBI (module) 271
DateTime::Format::Human (module) 271
DateTime::Format::Human::Duration (module) 271
DateTime::Format::Natural (module) 271
DateTime::Format::Pg (module) 268
DateTime::Format::SQLite (module) 271
DateTime::Format::Strptime (module) 270, 271
DB2 204
DBD::Gofer (module) 205
DBD::mysql (module) 206
DBD::mysqlPP (module) 204
DBD::Nagios (module) 205
DBD::Oracle (module) 206
DBD::Pg (module) 206
DBD::PgPP (module) 204
DBD::Proxy (module) 205
DBD::Wire10 (module) 204
DBD::WMI (module) 204
DBI 203, 349
connexion, 205
déconnexion, 209
erreur, 216
POE, 348
profiling, 219
requête
combinaison, 214
donnée de retour, 212
exécution, 211
liaison, 211
préparation, 209
test de connexion, 208
trace, 217
DBIx::Class
architecture, 233
schéma, 234, 237
DBI (module) 204–219, 224, 239, 349
DBI::Prof (module) 221
DBI::PurePerl (module) 204
$DBI_DRIVER (variable prédéfinie) 206
$DBI_DSN (variable prédéfinie) 206
$DBI_PROFILE (variable prédéfinie) 219, 220
$DBI_TRACE (variable prédéfinie) 217
DBIx::Class (module) 233, 236
DBIx::Class::Row (module) 234
DBIx::Class::Schema::Loader (module) 236

DBIx::Class::Schema::-Loader::Base (module) 236
DBIx::Connect::FromConfig (module) 207
déclaration
  de fonction, 26
  simple d'attribut Moose, 153
defined (fonction) 56, 78
délégation de méthode parente
  Moose 179, 180
delete (fonction) 78
delta_days() (fonction) 268
delta_md() (fonction) 267
delta_ms() (fonction) 268
déréférencement
  d'attribut Moose, 151
  de hash, 84
descripteur de fichier
  écriture, 187
  fermeture, 188
  lecture, 185
destruction d'objet Moose 159
die (fonction) 42, 184, 195, 341
do (fonction) 303
DOM (Document Object Model) 274, 284
DTD (Document Type Definition) 285
durée de temps 252, 263
  maintenant, 254

E

each (fonction) 83
Eclipse 4, 6, 16
écriture de fichier 187
ed 96
Editplus 5
élément
  de tableau, 51
  XML, 290, 291
Emacs 5
envoi d'événement POE 332
EPIC 4, 16
eq (opérateur) 31
espace de nommage 43
eval (fonction) 98, 303, 361
événement POE 327
exécuter perl 3
  fichier exécutable, 15
  ligne de commande, 13
  mode débug, 15
  sur un fichier, 15
exécution de requête SQL 211
exists (fonction) 56, 78
exit (fonction) 341
expression régulière
  alternative, 121
  ancre
    correspondance globale, 109
    début de chaîne, 108
    début de ligne, 106
    fin de chaîne, 108
    fin de ligne, 107
    frontière de mot, 108
    par préfixe, 109
  `Regexp::Assemble`, 126
  atome, 102
  classe de caractère, 104
  `Regexp::Common`, 124
  découpage, 122
  groupe
    capturant, 116
    non capturant, 119
  métacaractère, 102
  modificateur
    casse, 99
    correspondance globale, 101
    extension de syntaxe, 101
    ligne simple, 100
    multiligne, 100
  quantifieur
    *, 111

quantifieur (suite)
+, 112
?, 112
{n,m}, 113
{n}, 113
non avide, 114
possessifs, 115
recherche, 96
et remplacement, 97
stockage, 98
`Text::Match::Fast-Alternatives`, 128
`YAPE::Regex::Explain`, 128

F

fermeture de fichier 188
fichier
.ini, 320
de configuration, 320
écriture, 187
fermeture, 188
lecture, 185
ouverture, 183
`Path::Class`, 190, 191
Path : :Class
chemin, 189
temporaire, 197
`File::HomeDir` (module) 200
`File::HomeDir` (module) 199, 200
`File::pushd` (module) 202
`File::Slurp` (module) 187
`File::stat` (module) 193
`File::Temp` (module) 197, 198, 202
fonction
déclaration, 26
passage de paramètres, 27
valeur de retour, 28
variable locale, 29
`foreach` (mot-clé) 36
`foreach` (fonction) 82, 87
format de date 268
formulaire
HTML, 411
HTTP, 421

G

`ge` (opérateur) 31
génération de document XML 289
gestion d'erreur SQL 216
gestionnaire d'événement
`HTML::Parser` 366
`Getopt::Long` (module) 42
`grep` (fonction) 86, 89
grep 96
groupe d'expression régulière
capturant, 116
non capturant, 119
`gt` (opérateur) 31
`Gtk2::WebKit::Mechanize` (module) 426

H

hash 19, 74
affichage, 82
boucle, 82, 83
création, 75
déréférence, 84
élément, 76
liste de clés, 81
liste de valeurs, 81
référence, 84
suppression d'élément, 78
test d'élément, 78
tranche, 79
héritage de classe Moose 154
heure 261

HTML 114, 273, 363
  analyse, 363
    HTML::Parser, 365
    HTML::TokeParser, 377
    HTML::TreeBuilder, 383
    regexp, 364
  formulaire, 411
HTML::Element (module) 383, 386
HTML::Form (module) 411–413, 421
HTML::Form::Input (module) 412, 413
HTML::Parser (module) 276, 365–367, 377, 379, 381, 383, 385, 387
HTML::TokeParser (module) 376–379, 387
HTML::Tree (module) 383
HTML::TreeBuilder (module) 383, 384, 387
HTTP 389
  adresse, 390
  cookie, 410
  LWP, 395
    contenu, 396
    get, 395
    head, 396
    simple, 395
  LWP::UserAgent, 397
    contenu, 400
    création, 397
    get, 399
    head, 401
    post, 402, 403
    réponse, 398
  réponse, 393
  requête, 391, 392, 396, 399, 401-403
HTTP::Cookies (module) 398, 410–411
HTTP::Headers (module) 392, 399
HTTP::Message (module) 392–394, 399
HTTP::Reponse (module) 399
HTTP::Request (module) 393, 399, 402, 403, 413
HTTP::Request::Common (module) 402
HTTP::Response (module) 394, 397–401, 403, 407, 417

I

IDE 4, 13, 16
IDE 4
if (opérateur) 184
index (fonction) 26
Ingres 204
INI (format) 320
installer Perl
  sous Linux, 5
  sous Mac, 6
  sous Windows, 6
installer un module
  avec cpan, 10
  avec cpanm, 11
interprétation
  de date, 249
    DateTime, 270
    langue, 251
    str2time, 250
    strptime, 250
  de durée, 253
intervalle 48
  de temps, 252, 263
    maintenant, 254
inversion
  de liste, 60
  de tableau, 60
IO::File (module) 183–185, 188, 194
IO::Handle (module) 286
IO::Select (module) 326

IO::Socket (module) 326

J

JavaScript::V8 (module) 427
JE (module) 427
join (fonction) 25
jour 260
JSON (module) 310
JSON (JavaScript Object Notation) 308, 323
  sérialisation de données, 310
JSON::XS (module) 310

K

keys (fonction) 81
KiokuDB
  administration, 242
  récupération, 240
  stockage, 240
KiokuDB (module) 238–240, 242, 245
KiokuDB::Cmd (module) 242

L

last (mot-clé) 36
le (opérateur) 31
lecture de fichier 185
length (fonction) 54
lex 110
lier une requête SQL 211
List::MoreUtils (module) 47, 89
List::Util (module) 47, 89
liste 47
  aplatissement, 61
  de mots, 49
  de répertoires, 193
  dédoublonnage, 94
  inversion, 60
  mélange, 92
  plus grand élément, 90
  plus petit élément, 90
  premier élément, 89
  réduction, 91
  somme, 92
  tricotage, 93
local (fonction) 187
lt (opérateur) 31
LWP 395
  contenu, 396
  get, 395
  head, 396
  simple, 395
LWP (module) 389, 392–395, 397, 411
LWP::ConnCache (module) 398
LWP::Simple (module) 395
LWP::UserAgent (module) 397–399, 403, 404, 408–410, 413, 415, 417, 426
LWP::UserAgent 397
  contenu, 400
  création, 397
  get, 399
  head, 401
  post, 402
  réponse, 398
  requête, 403

M

m// (regexp) 96–98
MacPorts 6
maintenant 258
map (fonction) 86, 87, 89
Marpa (module) 110
Memcached 243
message différé POE 339

métacaractère d'expression régulière 102
méthode
d'un objet, 133
obligatoire de rôle Moose, 158
parente Moose, 176
modificateur d'expression régulière
casse, 99
correspondance globale, 101
extension de syntaxe, 101
ligne simple, 100
multiligne, 100
modification
d'accesseur Moose, 141
d'attribut Moose hérité, 156
de méthode Moose, 171-174
module
création, 43
utilisation, 44
mois 260
MongoDB 240
Moose 137
attribut
avec déclencheur, 150
chaînage, 153
construction, 147
déclaration, 138
déclaration simple, 153
déréférencement, 151
héritage, 156
obligatoire, 144
paresseux, 148
prédicat, 143
référence faible, 152
typage, 145
valeur par défaut, 145
classe
déclaration, 137
héritage, 154
constructeur, 159
héritage
attribut, 156
augmentation de méthode, 177
délégation de méthode, 179-180
super, 176
surcharge, 155
méthode
augmentation, 177
délégation, 179, 180
intercalage, 174
modification, 171
parente, 176
post-traitement, 173
prétraitement, 172
surcharge, 155
objet
accès, 140
accesseur, 141
construction, 159
déstruction, 159, 161
rôle
consommation, 157
création, 156
méthode obligatoire, 158
typage
création, 166
de base, 163
énumération, 167
personnalisé, 165
sous-type, 165
transtypage, 168
union, 167
`Moose` (module) 131, 135, 137-180, 238, 355
`Moose::FollowPBP` (module) 142
`Moose::Meta::Attribute::Native::Trait::Hash` (module) 181
`Moose::Object` (module) 138, 159
`Moose::Role` (module) 157

Moose::Util::TypeConstraints (module) 165, 166
MooseX::ChainedAccessor (module) 153
MooseX::Has::Sugar (module) 153
MooseX::POE (module) 355, 356
MooseX::SemiAffordance-Accessor (module) 142
MooseX::Singleton (module) 138
Mozilla::Mechanize (module) 426
my 19
MySQL 204

N

navigation web 407, 415
  authentification, 408
  formulaire, 411
    champ, 412
    validation, 413
  traitement des erreurs, 407
ne (opérateur) 31
next (mot-clé) 36
non-SQL
  clé-valeur, 243
  document, 244
notepad++ 5
noyau POE 329

O

open() (fonction) 284
opérateurs
  and, 34
  conditionnels, 32
  de chaînes, 25
  de test, 29, 31
  or, 34
  précédence, 25, 429
  sur scalaires, 24
  test négatif, 33, 34
Oracle 204
ORM (Object-Relational Mapping) 232
ouverture de fichier 183, 194

P

package (mot-clé) 43
Padre 4, 6, 16
paramètre
  d'événement POE, 333
  de fonction, 27
  ligne de commande, 41
  par référence, 39
parcourir un répertoire 196
parcours
  DOM, 277
  SAX, 281
  XPath, 280
Parse::RecDescent (module) 110
Parse::Yapp (module) 110
Path::Class (module) 189-191, 193, 195-196, 200, 201
  chemin, 189, 190
  fichier
    ouverture, 194
    suppression, 196
  répertoire
    création, 193
    information, 193
    liste, 193
    parcours, 196
    parent, 191
    sous-répertoire, 191
    suppression, 193

Path::Class::Dir (module) 191
Path::Class::File (module) 194
Path::Class:File (module) 191
POE (module) 326–362
POE (Perl Object Environment) 325
  composant
    DBI, 348
    de bas niveau, 348
    de haut niveau, 346
  distribué, 357
    client, 359
    serveur, 358
  entrée-sortie, 345
  événement, 327
    paramètre, 333
  message
    à heure dite, 340
    différé, 339
  noyau, 329
  principe, 326
  session, 328
    communication, 336
    création, 331
    variable, 335
  terminer, 341
  traitement, 341
POE::Component (module) 346
POE::Component::Client::-CouchDB (module) 244
POE::Component::EasyDBI (module) 348, 349, 354, 355
POE::Component::IKC::Client (module) 359
POE::Component::IKC::Server (module) 358
POE::Kernel (module) 329
POE::Wheel (module) 347
POE::Wheel::Run (module) 349
pop (fonction) 57
POSIX 96, 105
PostgreSQL 204, 239
précédence 25, 429
préparation de requête SQL 209
print (fonction) 20
print (mot-clé) 20
printf (fonction) 188
profiling SQL 219
programmation événementielle 325
  client, 359
  composant
    DBI, 348
    de bas niveau, 348
    de haut niveau, 346
    de niveau intermédiaire, 347
  distribuée, 357
  entrée-sortie, 345
  événement, 327
    envoi, 332
    paramètre, 333
  message
    à heure dite, 340
    différé, 339
  noyau, 329
  principe, 326
  serveur, 358
  session, 328
    communication, 336
    création, 331
    variable, 335
  terminer, 341
  traitement, 341
programmation objet
  classe, 132
  concept, 131
  constructeur, 135
  création d'objet, 132
  méthode
    appel, 133
    définition, 133

Moose, 137
constructeur, 159
héritage, 154
push (fonction) 57

Q

q() (opérateur) 21
QED 96
qr// (regexp) 99
quantifieur d'expression régulière
*, 111
+, 112
?, 112
{n,m}, 113
{n}, 113
non avide, 114, 115
qw (opérateur) 49

R

Redis 240, 243, 244
référence 18
accès, 38
déréférence, 38, 68
fonction, 40
hash, 84
paramètre, 39
scalaire, 37
tableau, 67, 69
regexp *voir* expression régulière
Regexp::Assemble (module) 126, 127
Regexp::Common (module) 124, 125
Regexp::Grammars (module) 110
Regexp::Keep (module) 109
répertoire
changement, 201
courant, 201
parent, 191
personnel, 199
temporaire, 198
réponse HTTP 393
requête HTTP 391, 392, 396, 399, 401–403, 417
return (mot-clé) 28
reverse (fonction) 60
rindex (fonction) 26
rôle Moose 156

S

s/// (regexp) 97, 98, 109
SAX (Simple API for XML) 275, 281
say (fonction) 20
say (mot-clé) 20
scalaire 18, 19
affichage, 19
déclaration, 19
déréférence, 38
initialisation, 19
opérateurs, 24
de chaînes, 25
référence, 37
test de définition, 26
scalar() (mot-clé) 24
SciTE 5
seconde 261
sérialisation de données 301
Data::Dumper, 301
JSON, 310
Storable, 307
YAML, 316
session POE 328
set_inner_xml (fonction) 293
shift (fonction) 58
sigil 18, 21, 38, 50
sin (fonction) 24
sous-répertoire 191

`splice` (fonction) 58
`split` (fonction) 25, 122-124
SQL
abstraction, 223
combinaison, 214
donnée de retour, 212
erreur, 216
exécution, 211
liaison, 211
ORM, 234
préparation, 209
POE, 348
profiling, 219
trace, 217
SQLite 204, 239
`Storable` (module) 307, 308
`strict` (module) 138
structure hybride 85
`substr` (fonction) 26
supprimer
fichier, 196
répertoire, 193
surcharge de méthode Moose 155
`system` (fonction) 42

T

table de hachage 18, 19, 74
affichage, 82
boucle, 82, 83
création, 75
déréférence, 84
élément, 76
liste de clés, 81
liste de valeurs, 81
référence, 84
suppression d'élément, 78
test d'élément, 78
tranche, 79
tableau 18, 50
à plusieurs dimensions, 67, 70, 71
affectation d'élément, 52, 55
affichage, 71
aplatissement, 61
boucle, 65
début, 58
déréférence, 68
dernier élément, 53
élément, 51
fin, 56
inversion, 60
milieu, 58
premier élément, 53
référence, 67, 69
suppression d'élément, 60
taille, 54
test d'élément, 55
tranche, 62
Tera Term 13
terminer un programme POE 341
`Text::Match::Fast-Alternatives` (module) 128
TextMate 5
`Tie::Hash::NamedCapture` (module) 119
`Time::Duration` (module) 252-255
`Time::Duration::fr` (module) 255
`Time::HiRes` (module) 340
Tokyo Cabinet 243
trace SQL 217
tranche de tableau 62
transtypage Moose 168
typage Moose 163, 165-168
types de données 18
liste, 47
structure hybride, 85
table de hachage, 19, 74
tableau, 18, 50

U

uc (fonction) 87
undef (mot-clé) 19, 23, 30, 385
Unicode 110, 311
unshift (fonction) 58
URI (module) 390, 391, 395, 399
URI (Uniform Resource Identifier) 363, 390
URI::http (module) 391
URL (Uniform Resource Locator) 390
use (mot-clé) 44
UTC (Temps universel coordonné) 262

V

valeur de retour 28
valeurs de hash 81
validation de formulaire HTML 413, 423
values (fonction) 81
variable
  de session POE, 335
  scalaire, 19
version de Perl 1
vim 5

W

warnings (module) 138
Web 389
  adresse, 390
  cookie, 410
  LWP, 395
    contenu, 396
    get, 395
    head, 396
    simple, 395
  `LWP::UserAgent`, 397
  navigation, 407
    authentification, 408
    formulaire, 411
    traitement des erreurs, 407
  réponse, 393
  requête, 391, 392, 396, 399, 401-403, 417
  `WWW::Mechanize`, 415
    authentification, 420
    back, 417
    cookie, 420
    création, 415
    formulaire, 421
    get, 417
    lien, 418
    post, 417
    rechargement de page, 418
    simple, 415
    traitement des erreurs, 420
while (mot-clé) 36, 37
Win32::IE::Mechanize (module) 426
WWW::Mechanize (module) 415-427
  back, 417
  cookie, 420
  création, 415
  formulaire, 421
    raccourcis, 423
    remplissage, 422
    sélection, 421, 422
    validation, 423
  get, 417
  lien, 418
  post, 417
  rechargement de page, 418
  traitement des erreurs, 420
WWW::Mechanize::Firefox (module) 427
WWW::Mechanize::Link (module) 419
WWW::Scripter (module) 427

X

XHTML 373
XML 273
  XML::LibXML, 275
    chargement, 275
    DOM, 277
    SAX, 281
    XPath, 280
  XML::Twig
    chargement, 286
    copier-coller, 297
    création, 285
    génération, 289
    handlers, 287
XML::LibXML (module) 275–277, 279–281, 286
XML::SAX (module) 281
XML::SAX::Base (module) 282
XML::SAX::Machines (module) 283, 284
XML::SAX::Manifold (module) 284
XML::SAX::Pipeline (module) 284
XML::Twig (module) 284–291, 294, 297, 299
XML::Twig::Elt (module) 285, 287, 289, 291–294
xor (opérateur) 73
XPath 280, 287, 295

Y

yacc 110
YAML 224, 313, 316, 323
YAML (module) 316
YAML::Syck (module) 316
YAML::XS (module) 316
YAPE::Regex::Explain (module) 128

LE GUIDE DE SURVIE

Perl moderne

Ce *Guide de survie* est l'outil indispensable pour programmer en Perl *aujourd'hui*. Il présente les dernières évolutions de Perl 5 par ses versions 5.10 et 5.12, fortement empreintes de la version 6 en cours de finalisation.

CONCIS ET MANIABLE

Facile à transporter, facile à utiliser — finis les livres encombrants !

PRATIQUE ET FONCTIONNEL

Plus de 350 séquences de code pour répondre aux situations les plus courantes et exploiter efficacement les fonctions et les bibliothèques d'un langage qui s'est radicalement modernisé.

Sébastien Aperghis-Tramoni, **Philippe Bruhat**, **Damien Krotkine** et **Jérôme Quelin** sont activement impliqués dans la communauté Perl française, *via* l'association **Les Mongueurs de Perl**, qui organise notamment « Les Journées Perl ». Ils sont les auteurs de nombreux modules Perl.

Niveau : Intermédiaire / Avancé
Catégorie : Programmation

PEARSON
Pearson Education France
47 bis rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4164-8
9 782744 041648