Programación orientada a objetos en Perl

Gunnar Wolf — gwolf@gwolf.org

http://www.gwolf.org/soft/poo_perl

Resumen

Perl es uno de los lenguajes favoritos dentro del mundo del software libre. Esta popularidad en buena parte se debe a lo fácil que es hacer todo tipo de tareas con Perl. Como dice uno de sus lemas, "Perl hace fáciles las cosas simples, y posibles las cosas complejas". Ahora bien, Perl se mantiene a ojos de mucha gente como un lenguaje demasiado limitado para tareas complejas.

La programación orientada a objetos es una poderosa técnica que nos permite desarrollar, organizar y redistribuir nuestro código mucho más limpiamente.

Mucha gente cree que Perl no es un lenguaje con el que se pueda hacer programación orientada a objetos - Esto es en buena medida porque Perl no impone ningún paradigma al programador. Perl, sin embargo, proporciona todo lo necesario para utilizarla.

En este tutorial veremos una introducción a qué es la programación orientada a objetos, cómo son y cómo se manejan los objetos desde Perl. Asumo que los asistentes tengan al menos buenos conceptos de programación en Perl.

Índice

Licencia de uso, modificación y redistribución

1. Introducción a los objetos

La programación orientada a objetos es una metodología que desde mediados de los 80s se ha vuelto muy popular, gracias principalmente a los lenguajes C++ y Smalltalk, y es manejada por prácticamente todos los lenguajes modernos de propósito general — Incluso, muchos de estos lenguajes (y tal vez el mejor ejemplo sea Java) exigen que toda la lógica del programa se maneje utilizando el paradigma orientado a objetos. Pero no nos adelantemos, vamos a ver antes qué es la Programación Orientada a Objetos (a la que a partir de ahora me referiré como POO). Cubro en las próximas secciones las principales características de

los objetos — Claro está, siendo esto una introducción, algunos de los conceptos los explico con mayor detenimiento, mientras que otros (los menos importantes para comprender los conceptos) serán brevemente mencionados.

Para los ejemplos en esta sección utilizaré pseudocódigo, para ilustrar de la manera más clara e independiente de filosofías linguísticas los conceptos.

De antemano: La POO puede parecer muy compleja a primera vista, y mucha gente —claro está, yo incluído— le huye. En este tutorial les voy a presentar los conceptos y la manera de implementarlos en Perl. La POO, sin embargo, no es muy apta para ser enseñada en simples tutoriales, pues donde mejor se aplica es en proyectos grandes. ¿Mi recomendación? Sigan este tutorial o cualquier otro texto, aprendan lo que es... Y practiquen, lean código, participen en proyectos grandes. De repente, ni cuenta se darán y ya tendrán los concpetos asimilados.

1.1. Definiciones

Comencemos definiendo tal como lo hace Damian [3, 4], para tener un poco de vocabulario común antes de lanzarnos a trabajar.

Hay cinco cosas básicas a aprender para comprender la POO:

- Un *objeto* es cualquier cosa que me permite localizar, utilizar, modificar y asegurar datos.
- Los *atributos* son características (datos) que tiene un objeto
- Una *clase* es una descripción de qué datos pueden utilizare a través de un tipo de objeto, y cómo puede hacerse.
- Un *método* es la manera en que puedo utilizar (consultar, modificar o procesar) los datos de un objeto.
- La *herencia* es el mecanismo que permite que amplíe las clases existentes para que provean datos o métodos adicionales.
- El *polimorfismo* es la manera en que diferentes objetos responderán de diferente manera al mismo mensaje, dependiendo de la clase a la cual pertenezcan.

Tanto los métodos como los atributos pueden ser *de clase*, si son concernientes a todos los objetos que forman parte de la clase, o *de instancia*, si son concernientes a un objeto en específico.

Hay varios conceptos adicionales relativos a los objetos que son fundamentales en algunos lenguajes, como la noción de métodos y atributos *privados*, *públicos* y mezclas entre ambos. En Perl, al igual que en otros varios lenguajes, no creemos en esta discriminación, así que no ahondaré al respecto. Hay varios textos (como [5]) abundando en el tema y mostrando cómo implementarlo en Perl.

¿Suena confuso? No se preocupen, vamos a los ejemplos. Y si aún eso no basta, un poco de lectura filosófica siempre puede ayudar[1]

[1] Y lo digo completamente en serio. Yo entendí mucho mejor la POO tras leer la obra

1.2. Ejemplificando con pelos y señales

Muchos problemas, así como los componentes de muchos problemas, pueden ser vistos como análogos de un objeto en la vida real — pongamos como ejemplo a Tin Tan, el gato que me acompaña en esta fotografía:

Tin Tan tiene ciertos atributos — Pesa aproximadamente cinco kilos, es macho, tiene pelo espeso y corto, y es hijo de Chupchic y Sandunga (quienes, sobra decirlo, son también gatos). Puede hacer varias cosas — Puede comer, dormir y jugar. Puede tener hambre o sueño, cosas que representaremos como acciones. Incluso puedo hablar de atributos suyos como cosas no físicas, como si ya comió, de qué humor está o en qué está fijando su atención. Hay incluso detalles que no le importan a quien trate con él, pero para él son fundamentales — Por ejemplo, si comió demasiado seguramente va a sentirse mal y vomitar un poco.

La POO consiste en definir el problema a resolver con nuestro programa como interacciones entre una serie de objetos. Vamos, pues, a definir a Tin Tan utilizando pseudocódigo, y más tarde lo definiremos en Perl.

Vamos a definir antes que nada a un gato genérico:

```
01 clase Gato [
02  atributos_inst: peso, pelo, sexo, padre(Gato.sexo='M'),
03    madre(Gato.sexo='F'), lleno, sueño;
04  atributos_clase: cantidad
05
06    creación: lleno=0, sueño=0, cantidad=cantidad+1;
```

de Santo Tomás de Aquino [6]. Muy en resumen: Al hablar respecto a los ángeles, ¿qué los diferencía de nosotros? Que los ángeles son pura esencia, no son corpóreos, y cada uno de ellos es una especie única. No son corruptibles, pues la corrupción implica modificar características de un individuo respecto a su especie, y cualquier característica que se modificara en un ángel modificaría a la especie entera. Por otro lado nosotros, los seres humanos, somos seres corpóreos y pertenecemos a una misma especie. Nos distinguen a unos de otros nuestros *accidentes concomitantes*, nuestras imperfecciones y lo que éstas generan. Los ángeles, por tanto, son *clases*, y todos sus atributos y métodos deben ser *de clase*. Nosotros tenemos varios comportamientos y características inherentes a todos los seres humanos —nuestros métodos y atributos de clase— pero del mismo modo tenemos comportamientos y características particulares a cada uno de nosotros —nuestros métodos y atributos de instancia.

No es un paralelo perfecto, pero ayuda a comprender estos conceptos.

```
07
08    metodo come (cuanto_ingirio) [
09        lleno = lleno + cuanto_ingirio;
10        si (lleno > peso / 10) [ vomita; ]
11        sueño = sueño + 2;
12    ]
13
14    metodo duerme [
15 si (sueño < 1) [ return 'No tengo sueño'; ]
16 sueño = 0;
17    ]
18
19    metodo juega [
20      si (tiene_sueño) [ return 'Estoy muy cansado'; ]
21      si (tiene_hambre) [return 'Tengo hambre'; ];
22      lleno = lleno - 1;
23      sueño = sueño + 1;
24    ]
25
26    metodo vomita [ lleno = lleno * 2/3; ]
27
28    metodo tiene_hambre [
29        si (lleno < 5) [ return 1; ] else [ return 0; ]
30    ]
31
32    metodo tiene_sueño [
33        si (sueño > 10) [ return 1 ] si_no [ return 0; ]
34    ]
35
36 ]
```

En la primer línea vemos que estamos creando una nueva clase llamada Gato.

Definimos los atributos en las líneas 2 a 6. En este pseudolenguaje, al definir un atributo como `padre(Gato.sexo='M')` indico que el valor que ocupe el atributo `padre` debe ser un `Gato` con sexo `'M'`, y con `madre(Gato.sexo='F')` declaro que `madre` debe ser un `Gato` con sexo `'F'`. Además, indico que en el momento de la creación de un nuevo gato, este *nacerá* con hambre y sin sueño, y la cuenta total de gatos que tengo se incrementará en uno.

La mayor parte del cuerpo de la clase típicamente es la definición de métodos. Acá vemos los métodos `come`, `duerme`, `juega`, `tiene_hambre`, `tiene_sueño` y `vomita`, en las líneas 8 a 34.

Con esto definido, podemos ya crear y usar a nuestro gato. Veamos pues:

```
01 Tin_Tan := Gato;
02 Tin_Tan.peso = 5;
03 Tin_Tan.pelo = 'espeso y corto';
```

```
04 Tin_Tan.sexo = 'M';
05
06 Chupchic := Gato;
07 Chupchic.sexo = 'M';
08 Sandunga := Gato;
09 Sandunga.sexo = 'F';
10
11 Tin_Tan.padre = Chupchic;
12 Tin_Tan.madre = Sandunga;
13
14 mientras (Tin_Tan.tiene_hambre) [ Tin_Tan.come; ]
15 Tin_Tan.juega
16 Tin_Tan.juega
17 si (Tin_Tan.tiene_sueño) [ Tin_Tan.duerme; ]
```

Claro, es un ejemplo muy simple, pero podemos ver la lógica básica de la POO: Le estamos indicando las acciones a realizar a `Tin_Tan`, que es un objeto o una *instancia* de la clase `Gato`, y esta tiene asociadas a sí sus funciones y métodos.

1.3. La herencia de mis ancestros

La POO nos permite, a través de la herencia, ahorrar código. Vamos a poner un ejemplo: Los gatos, así como perros y las moscas, son animales. Todos los animales comparten características comunes: Todos duermen y comen. Todos tienen un padre y una madre. Podríamos entonces definir una clase `Animal`, que definiera las características y métodos básicos, y permitiera a los diferentes animales hacer cambios únicamente donde hiciera falta. La clase `Animal` sería la *clase base* o *clase padre*, mientras que las clases `Gato`, `Perro` y `Mosca` son sus *clases derivadas* o *clases hijas*.

Cuando hablamos de herencia, es común usar la nomenclatura *is-a* (*es-un*). Un objeto instancia de `Gato`, por pertenecer a la clase `Gato`, *es-un* `Animal`, y todas las operaciones que podemos realizar con un `Animal` debemos poder realizarlas también con un `Gato`.

Para no aburrirlos con pseudocódigo, veremos ejemplos de esto ya cuando estemos hablando de Perl.

1.3.1. Herencia multinivel

La herencia, claro, no se limita a un nivel. Al igual que en el mundo real, un `Gato` es un `Animal`, pero un `Animal` es un `Ser_vivo`, y —aunque muchos opinen lo contrario, es lo más indicado para este ejemplo— un `Ser_vivo` es una `Cosa`. Y básicamente todo lo que veamos puede ser representado como una `Cosa`, la clase más genérica que hay, que en realidad no implementa ningún método ni tiene ningún atributo.

1.3.2. Herencia múltiple

En muchos lenguajes (aunque no en todos), una clase puede heredar de más de una, permitiendo que en vez de un árbol de herencias haya una malla de herencias. En general, es recomendable mantener una maraña lo menos densa posible, tendiente a un árbol, pero es a fin de cuentas el programador quien decidirá cómo implementarlo.

Para amarrar nuestro ejemplo a la herencia múltiple, un Gato puede ser muchas cosas además de un animal — Por ejemplo, un gato puede verse como un excelente `Entropizador`[2]. Y hay muchas clases que pueden ser derivadas de Entropizador, como una `Reunion_social`, un `Bebé` o nuestro amigo el `Tiempo`. Un Gato, entonces, sería tanto un `Animal` como un `Entropizador`.

Hay bastante debate entre los teóricos de la orientación a objetos respecto a si la herencia múltiple es buena o mala, pues, a pesar de darnos un modelo mucho más completo del mundo, puede hacer la programación bastante más compleja. Algunos lenguajes —notablemente Java— han decidido no implementarla, decantándose por las *interfaces*. Varios lenguajes la implementan, siempre que ninguna de las clases o sus superclases implementen métodos o atributos con el mismo nombre (o permitiendo renombrar a los que entran en conflicto, como lo hace Eiffel). Perl, pragmáticamente como siempre, permite la herencia múltiple, y en caso de haber conflictos, los resuelve usando el primer nombre que aparezca en una búsqueda por profundidad (*depth-first*) sobre las clases en el órden en que fueron declaradas en `@ISA` — Aunque mejor no corremos, y dejamos esa discusión para cuando llegue el momento (subsección 1.3.2).

1.3.3. El todo es mayor que la suma de sus partes: *tiene-un*

Una clase puede heredar a otra, como lo mencionamos en esta sección. Una clase también puede, por otro lado, especificar que sus objetos incluyan objetos de otras clases, implementando una relación *tiene-un*. Por ejemplo, si tenemos ya una clase implementando el control de una `Puerta`, probablemente queramos definir a nuestra clase casa de modo que uno de sus atributos sea un objeto `Puerta`. Sería incorrecto buscar que la casa heredara el comportamiento de una puerta — Simplemente, la casa no *es-una* puerta. Más bien, para definir mi casa:

```
mi_casa := Casa;
mi_casa.puertaPrinc := Puerta;
mi_casa.puertaPrinc.tipo = 'Metal';
mi.casa.puertaPrinc.abre;
```

Esta característica *no es* herencia. La clase `Casa` simplemente *incluye* una puerta en el lugar adecuado para ser utilizada del modo que lo esperamos. La clase `Casa` no requiere acceso especial a `Puerta`, no cambia ninguno de sus

[2] ¿Un entropizador? Sí, un efectivísimo agente generador de entropía.
Es relativamente simple tener una casa limpia y en órden si no hay mascotas. Es complicado tener una casa limpia y en órden con un gato. Es casi imposible lograrlo, como en mi caso, cuando aumenta la cantidad de gatos.

comportamientos — Es muy común y cómodo ver este tipo de *agregación* entre objetos.

1.4. Polimorfismo

Algo estrechamente relacionado con la herencia es el *polimorfismo*: Diferentes maneras de llevar a cabo una acción, un método que responde de manera diferente dependiendo de cómo o desde dónde fue llamado. Por ejemplo, todos sabemos que las `Moscas` y los `Humanos`, a diferencia de casi todos los otros `Animales`, se tallan las manos antes de comer[3]. Entonces, en nuestra simplísima implementación, bastaría con redefinir el método `come` reflejando nuestro poco usual comportamiento.

A estas alturas, espero, ya comprendieron todos los conceptos básicos de la POO. Vamos, pues, a Perl.

1.5. Sobrecarga de operadores

Las operaciones que podemos hacer sobre los tipos de datos nativos en cualquier lenguaje —suma, resta, concatenación, y un largo etcétera— muchas veces quisiéramos tenerlas disponibles para nuestros objetos. Sin embargo, si yo le digo a algún lenguaje que sume `obj1 + obj2`, si no se queja y muere amargamente por ser esta una operación no definida, me sumará tal vez dos direcciones en memoria - y no su contenido.

La sobrecarga de operadores significa que en nuestra clase indicamos — normalmente de la misma manera que si definiéramos un método— cómo aplicar un operador del lenguaje a nuestra clase. Claro, nada evita que usemos el signo `+` para multiplicar, el `-` para sumar y el `*` para concatenar, pero —espero— ninguno de nosotros es lo suficientemente esquizofrénico como para intentar sabotearse a sí mismo de esa manera.

2. Revisión de conceptos en Perl

Aunque antes de entrar de lleno en la orientación a objetos en Perl, repasemos unos puntos que tienden a ser olvidados y son fundamentales para trabajar con objetos.

Tenemos que recordar que en Perl todas las variables y funciones, así como algunos otros tipos de datos, están representados en *tablas de símbolos*. Tenemos las siguientes tablas independientes en Perl. Los tres primeros apuntan a datos, mientras que los demás apuntan a otro tipo de estructuras:

`SCALAR` Pueden guardar un sólo valor (`$escalar`)

`ARRAY` Guardan una colección de valores indexados por posición (`@arreglo`, `$arreglo[2]`)

[3] ¿Que tú no lo haces? ¿Pero qué clase de animal eres?

`HASH` Guardan una colección de valores indexados por llave alfanumérica (`%hash`, `$hash{llave}`)

`CODE` Apuntan a código con nombre — a funciones (`&func()` o simplemente `func()`)

`FORMAT` Apuntan a definiciones de formato de salida (sin prefijo identificador — `format Formato = (...)[7]`)

`FILEHANDLE` Permiten el manejo de archivos o de directorios (sin prefijo identificador — `FH`)

`GLOB` En realidad no son un tipo de datos separado - se refieren a *cualquiera* de los tipos de datos anteriores[4] que tenga el mismo nombre después del prefijo identificador (`*glob` se refiere a `$glob`, `@glob`, `%glob`, `&glob`, el filehandle `glob` y el formato `glob`)

2.1. Ámbito de las variables

Cuando comenzamos a trabajar en Perl, podemos usar las variables conforme las vayamos necesitando, y se van creando en lo que parece un espacio universalmente disponible. Esto es muy cómodo para proyectos pequeños, pero puede rápidamente volverse inmanejable para proyectos más grandes, por las muy probables colisiones de nombres que se presentarían (por ejemplo, ¿en cuántos lugares del código podemos referirnos a la variable `$tmp`? ¿Qué tan probable es que de algún modo modifiquemos el valor de una variable que no queríamos?) Para resolver estos problemas, la primer gran e importante solución es la definición del *alcance* o del *ámbito* para nuestras variables.

Si no le indicamos nada a Perl, nuestras variables serán *globales*. Cuando creamos una variable, ésta estará disponible en cualquier parte de nuestro programa — Con todo lo bueno y malo que eso implica.

Podemos, por otro lado, confinar una variable a una región específica. Para hacer esto, declaramos a la variable como de *ámbito léxico*. La primera vez que utilicemos a nuestra variable, la *declaramos* como léxica con `my $var`, o podemos declarar como léxicas a un conjunto de variables con `my ($var1, @var2, %var3)`. Esto hará que las variables en cuestión nazcan después de su declaración y desaparezcan (o salgan de ámbito) al terminar el bloque de código en el que fueron declaradas. Las variables de ámbito léxico no pueden ser, además, accesadas desde ninguna otra función, aún si esta es invocada desde dentro del bloque. Vamos a un ejemplo pues:

```
$var1 = 1;  # Una variable normal, de ámbito global
$var2 = 10; # Otra variable global
trabaja();
print "En la base - var1=$var1, var2=$var2\n";
```

[4] Esto, sí, merecería una explicación amplia... Pero como no vamos a utilizarlos, y como son una fuente de preguntas muy prolífica, creo que lo mejor será ignorarlos temporalmente ;-)

```
juega();
print "En la base de nuevo - var1=$var1, var2=$var2\n";
sub trabaja {
    my $var = 4; # Es léxica
    my $var2 = 1; # Es léxica aunque lleve el nombre de una global
    $var1 = $var + $var2; # Esta sí es global
    $var2 = $var2 + 1; # Una vez declarada, siempre será léxica
    print "Tenemos var=$var, var1=$var1 y var2=$var2\n";
}
sub juega {
    my $var1 = $var1 * 2; # Primero evalúa con global, luego ve ámbito
    my $var2 = $var1 + $var2; # Mismo caso - pero $var1 ya es léxica
    print "Ahora tenemos var1=$var1 y var2=$var2\n";
}
__END__
Tenemos var=4, var1=5 y var2=2
En la base - var1=5, var2=10
Ahora tenemos var1=10 y var2=20
En la base de nuevo - var1=5, var2=10
```

Si utilizamos el pragma `strict` [9] en nuestros programas (lo cual es muy recomendable, pues nos evita caer en vicios y prácticas inseguras de programación), incluso las variables globales requieren ser declaradas con `our` *`variables`* o con `use vars qw(`*`variables`*`)` para ser utilizadas, o cuando menos calificadas con su espacio de nombres (aguanten, hablaremos de esto muy pronto).

Hay un ámbito adicional, que hasta hace unos años, en la era de Perl 4, era tan recomendado como el léxico (mismo que, claro, aún no existía): el ámbito *local*. Este ámbito, sin embargo, presenta muchos problemas y puede llevar a bugs bastante obscuros, por lo que está recomendado evitarlo — de hecho, si utilizamos el pragma `strict`, no podremos usar local aunque queramos, por nuestro propio bien. Hay ocasiones, claro, en las que local es la respuesta a nuestros problemas. A quien le interese leer más acerca de dichas ocasiones, le sugiero el artículo de Mark Jason Dominus [8].

2.2. Espacios de nombres, bibliotecas y módulos

Aún estas técnicas pueden no ser suficientes. Cuando trabajamos en algún proyecto grande, es importante poder definir unidades organizacionales para nuestro programa. Los espacios de nombres, además de esto, nos proporcionan el mecanismo básico para la definición de clases.

Todos los símbolos que utilicemos en Perl pertenecen a un espacio de nombres, aún si no lo hacemos explícito — `main`. Esto significa, básicamente, que a menos que indiquemos lo contrario, cuando usamos la variable `$datos` estamos en realidad utilizando `$main::datos`. Si creamos la función `busca_datos()`, en realidad estamos definiendo `$main::busca_datos()`. Referirnos a un símbolo por su nombre completo, valga mencionar, es conocido como *calificar* el símbo-

lo, o *utilizar el nombre calificado completo* del símbolo. ¿De qué nos sirve esto? De que, de la misma manera, podemos trabajar con símbolos que viven en `main`, podemos definir nuestros propios espacios de nombres — Por ejemplo, podríamos implementar las funciones `Gato::come()` y `Gato::Duerme`, que (volviendo al ejemplo que revisábamos hace unas secciones) nos servirían para que nuestro `Gato` coma y duerma. Si bien el usar los espacios de nombres definitivamente no implica que estemos programando con una metodología orientada a objetos, sin duda alguna nos ayuda a clasificar nuestros símbolos y evitar choques de nombres.

Ahora, sería muy engorroso tener que calificar cada uno de los símbolos que manejáramos. La instrucción `package` nos ayuda a manejar esto con mayor claridad. A partir de que una de estas instrucciones aparece, el espacio de nombres implícito se vuelve el que le indicamos a package. Vamos a verlo en un ejemplo:

```
$var = 20;
$valor = 25;
sub dime { return @_; }

package otro;
$var = $main::var / 2;
$valor = $var + 1;
sub dime { return 'Recibí: ', join(', ', @_); }

package uno_mas;
print "En main tengo $main::var y $main::valor\n";
print "Dime(1,2,3) me da: ", main::dime(1,2,3),"\n";
print "En otro tengo $otro::var y $otro::valor\n";
print "Dime(1,2,3) me da: ", otro::dime(1,2,3),"\n";
__END__

Resultado de la ejecución:

En main tengo 20 y 25
Dime(1,2,3) me da: 123
En otro tengo 10 y 11
Dime(1,2,3) me da: Recibí: 1, 2, 3
```

Normalmente utilizamos los espacios de nombres siempre en conjunto con bibliotecas o, mejor aún módulos — Archivos que contienen código listo para ser incluído en nuestro programa. Estos nos permiten empaquetar y reutilizar nuestro código fácilmente — y son fundamentales si queremos hacer programación orientada a objetos en serio.

Las bibliotecas son fragmentos de código guardados en un archivo. Este código normalmente consiste únicamente de definiciones, sin nada directamente ejecutable, pues normalmente en una biblioteca buscamos únicamente funciones disponibles. Para incluir una biblioteca, le indicamos a Perl `require`

`'biblioteca.pl';` y, en el momento que la ejecución llega a este punto, se ejecuta el código existente en el archivo[5]. Es importante señalar que normalmente las bibliotecas terminan con `1;` para asegurar que la última instrucción ejecutada en ella entregue un valor verdadero, pues de lo contrario la inclusión es tomada como fallida, y la ejecución del programa es abortada.

Las bibliotecas cada vez son menos utilizadas, sin embargo. Su reemplazo son los *módulos*. Las principales diferencias entre ellos es que:

- Los archivos que contienen módulos de Perl llevan la terminación `.pm`
- Los módulos típicamente incluyen el espacio de nombres concordante con su nombre de archivo. El módulo `Gato.pm` iniciará típicamente con la instrucción `package Gato;`
- Los módulos son incluídos con la instrucción `use` en vez de `require`. Esto implica que
  - Se ejecutan en tiempo de compilación, no cuando el flujo del programa llega a la instrucción (es equivalente a un `require` dentro de un bloque `BEGIN`)
  - No requiere utilizar comillas alrededor del nombre del módulo
  - Permite especificar una lista de símbolos a *importar* del módulo a nuestro espacio de nombres base. Un módulo no debe alterar el espacio de nombres de quien lo llama a menos que sea explícitamente solicitado.
- Como parte de su nombre pueden incluir los caracteres `::` lo que indica jerarquía en el directorio — El módulo `Gato::Peludo` se traduce al archivo Gato/Peludo.pm en alguno de los directorios referidos en `@INC`. El espacio de nombres correspondiente, sin embargo, es `Gato::Peludo` mismo — Sin implicar con esto que el espacio de nombres sea jerárquico. `Gato::Peludo` no forma parte de `Gato`, mas que visualmente. Los programadores humanos de todos modos usamos esa afinidad visual para representar jerarquías, lo cual es un uso completamente válido, aunque Perl no sepa nada al respecto.

2.3. Manejo de referencias

El esquema de símbolos existente en Perl, definido a lo largo de la historia del lenguaje básicamente hasta antes de Perl 5, es muy cómodo y flexible, pero tiene un par de inconvenientes:

- Al pasar argumentos no escalares a una función o guardarlos en una lista, estos son *aplanados* — Si tenemos `@arr = (1,2,'asdf')`, `$val =`

[5] El archivo en cuestión, en este caso `biblioteca.pl`, será buscado en los directorios que aparecen en `@INC`, en el órden en que aparecen

`'algo',%hash = (llave =>'valor')`, y llamar a la función `func(@arr, $val, %hash)` la lista de argumentos queda aplanada a un feo e inmanejable `(1, 2, 'asdf', 'algo', 'llave', 'valor')`

- Hay varios tipos de datos que no pueden ser pasados directamente (`CODE`, `FORMAT` y `FILEHANDLE`), y requieren el uso de `GLOB`s — lo cual muchas veces se vuelve poco claro.

Es en buena parte por esto que nace la noción de las *referencias*[6], pero su alcance llega mucho más allá de simplemente corregir estos problemas. Como veremos, Perl implementa los objetos usando referencias. Veamos pues un breve repaso de cómo se crean y utilizan las referencias en Perl.

Una referencia es un valor único que apunta a un tipo de datos determinado y, por consiguiente, se guarda en un valor escalar. La manera canónica de referirse a datos ya existentes es anteponerles un caracter \.

Ahora, siendo referencias, si pedimos el valor que guardan, nos entregan la dirección en memoria de los datos, lo cual en realidad no nos es nada útil. Para accesar a los datos referidos, podemos encerrar a la referencia entre llaves y anteponerle el símbolo del tipo de datos real o, si se trata de una llamada a función o de un elemento de un arreglo o hash, poner una flecha (->) entre el nombre de la variable y el indicador de subíndice o llamada. Cuando no hay lugar a ambiguedades, podemos omitir las llaves y las flechas.

La función interna de Perl `ref` nos indica a qué tipo de datos apunta una referencia (entrega una cadena indicándolo o una cadena vacía, en caso de no tratarse de una referencia). Va un par de ejemplos:

```
$escalar = 'valor';
@arreglo = (1,2,3);
%hash = (llave => 'valor', key => 'value');
sub test { my $a = shift; return "Probando - $a"; }

$ref_esc = \$escalar;
$ref_arr = \@arreglo;
$ref_hash = \%hash;
$ref_sub = \&test;

@arr_refs = ($ref_esc, $ref_arr, $ref_hash, $ref_sub);

@arr_multidim = ([1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]);

print "Los valores de las referencias mismas:\n";
```

[6]Bueno, seguimos sin poder pasar un `FILEHANDLE` a una referencia más que a través de un `GLOB`, pero hay un bonito módulo que es parte de la distribución oficial de Perl, `IO::File`, que nos proporciona una bonita interfaz orientada a objetos para el manejo de archivos

```
print "$ref_esc, $ref_arr, $ref_hash, $ref_sub\n";

print "\nLos verdaderos valores referidos:\n";
print join(', ', $$ref_esc, $ref_arr->[0], $ref_hash->{llave},
   $ref_sub->('parámetro')),"\n";

print "\nAccesando a elementos para los que sí hace falta desambiguar\n";
print join(', ', ${$arr_refs[0]}, ${$arr_refs[1]}[0], ${$arr_refs[2]}{key},
   $arr_refs[3]->('de nuevo')),"\n";

print "\nProcesando un arreglo multidimensional\n";
foreach $reng (0, 1, 2) {
    foreach $col (0, 1, 2) {
print $arr_multidim[$reng][$col];
    }
}

print "\n\nConsultando los tipos de datos referidos\n";
print join(', ', map {ref $_}
   $escalar, $ref_esc, $ref_arr, $ref_hash, $ref_sub),"\n";
__END__

Resultado de la ejecución:

Los valores de las referencias mismas:
SCALAR(0x8169318), ARRAY(0x8169348), HASH(0x81693a8), CODE(0x8169408)

Los verdaderos valores referidos:
valor, 1, valor, Probando - parámetro

Accesando a elementos para los que sí hace falta desambiguar
valor, 1, value, Probando - de nuevo

Procesando un arreglo multidimensional
123456789

Consultando los tipos de datos referidos
, SCALAR, ARRAY, HASH, CODE
```

Para una explicación a mayor profundidad de las referencias, sugiero consultar la página de perldoc perlref, el capitulo 8 de Programming Perl, los capítulos 1 y 4 de Advanced Perl Programming y el capítulo 11 del Perl Cookbook [10, 14, 15, 16]

3. Usando y creando objetos en Perl

Partiendo de acá, manejar los objetos en Perl resulta muy simple. Vamos desde el principio: Antes de aprender a crearlos, veamos cómo utilizar a los objetos.

Hay un punto que tenemos que agregar antes de continuar a lo que mencionábamos respecto a las referencias. Los objetos tienen *métodos*, funciones asociadas a todo miembro de determinada clase. Si tenemos un objeto `$obj` y queremos llamar a su método `dime` lo hacemos con `$obj->dime`. Y, consistentemente, cuando queremos llamar a un método de clase, lo hacemos anteponiéndole el nombre de la clase en vez del de un objeto: `Objeto->crea` llamará probablemente al método de clase de `Objeto` que se llama `crea`.

3.1. Utilizando objetos

Manejar un objeto significa saber crearlo, trabajar con él, solicitar/modificar sus atributos y, llegado el momento, destruirlo. Para ir siguiendo este ejemplo pueden querer asomarse a la clase Gato que crearemos poco más adelante, en la subsección 3.2.4:

3.1.1. Creación de un nuevo objeto

Si, como comentábamos, un objeto no es más que una simple referencia, podemos utilizar cualquier variable escalar (o cualquier elemento en una lista o hash) para guardarlos. Vamos a crear en Perl, utilizando una imaginaria clase `Gato` (que más adelante definiremos) a nuestro Tin Tan:

```
$tin_tan = Gato->new(-sexo=>'M', -peso=>5,
-pelo=>'espeso y corto');
```

Esto nos entrega un nuevo objeto `Gato` en nuestra variable `$tin_tan`, con las características que le especificamos. Es común que, como parte del constructor, revisemos si podemos crear un objeto tal como nos fue pedido, y en caso de no ser posible normalmente regresamos el valor `undef` en vez de un objeto. Por ello, normalmente a la creación del objeto sigue algo así:

```
warn 'No pude crear el Gato solicitado!' unless defined $tin_tan;
```

3.1.2. Uso de los métodos

El objeto creado nos permite llamar a sus métodos de esta manera:

```
while ($tin_tan->tiene_hambre) { $tin_tan->come; }
$tin_tan->juega; $tin_tan->juega;
if ($tin_tan->tiene_sueño) { $tin_tan->duerme; }
```

3.1.3. Acceso a los atributos

Les puede llamar la atención que los nombres de los atributos (las llaves del hash) inician con guión. Esto no es un requisito, sino que una convención —

Continuemos, pues, la tradición. Podemos acceder a los valores usándolos como llaves del hash referenciado:

```
$kilos = $tin_tan->{-peso};
```

Ahora bien, es común preferir dar acceso a los atributos a través de métodos *accesores* (para consultar el valor de un atributo) y *mutadores* (para modificar el valor de un atributo). Esto no siempre es indispensable, pero sí nos proporciona una buena medida adicional de encapsulamiento, evitando que el usuario de nuestra clase le dé a un atributo un valor no válido. Los accesores típicamente en inglés llevan el nombre `get_`*`atributo`* (en el ejemplo que muestro más adelante en español los llamo `lee_`*`atributo`*), y los mutadores `set_`*`atributo`* (y yo uso `pon_`*`atributo`*). Es común también el unir accesor y mutador en un sólo método llamado a secas *`atributo`* con un leve dejo de polimorfismo, pero yo les recomiendo seguir la recomendación de Damian Conway ([3] páginas 19 a 21) y, por simplicidad psicológica, manejarlos como métodos independientes.

Utilizando accesores y mutadores, pues, podemos hacer lo siguiente:

```
if ($tin_tan->lee_peso <4) {
warn 'Tu gato está demasiado flaco!';
}
if ($otro_gato->lee_sexo eq 'F') {
warn 'Más vale cuidar la convivencia con Tin Tan';
}
```

3.1.4. Destruyendo el objeto

Llega un momento en el flujo de todo programa en que ya no necesitamos más a los objetos que habíamos creado, y hay que destruirlos. Afortunadamente, Perl nos ayuda en este penoso proceso: No tenemos que hacer nada, Perl sabe perfectamente cuándo el objeto en cuestión puede ser destruído. Perl maneja un mecanismo de conteo de referencias en cada uno de los datos representados en la memoria, y cuando ese conteo es cero, sabe que puede disponer nuevamente de la memoria que ocupaba — y, en caso de los objetos, que llegó la hora de llamar al método `DESTROY`. De ese modo, si tenemos muchos gatos en casa y regalamos uno:

```
my $gatos_antes = Gatos->cantidad;
$gatita = undef;
my $gatos_ahora = Gatos->cantidad;
print "Yo tenía $gatos_antes, ahora tengo $gatos_ahora\n";
# Resultado: Yo tenía 15 gatos, ahora tengo 14
```

3.2. Qué va dónde

Como vimos previamente, todos los objetos que utilicemos son instancias de alguna clase. Resulta lógico que si queremos aprender a jugar con objetos debemos empezar definiendo una clase. En Perl para crear una clase usamos

los *espacios de nombres*[7] — Para indicar que vamos a crear la clase `Gato`, le indicamos a Perl `package Gato;`, y a partir de ese punto estaremos trabajando sobre la clase `Gato`. Las funciones que definamos serán los métodos de esta clase, los atributos de clase serán las variables globales que creemos dentro de ese espacio de nombres.

Típicamente cada clase será implementada en un archivo independiente, en un *módulo*. Este módulo por comodidad lleva por nombre el de la clase con el sufijo `.pm` (de *Perl Module*), y para ser incluído desde otro programa de Perl, debe estar ubicado en algún directorio que aparezca en `@INC`[8].

3.2.1. Benditas referencias

En Perl, siendo estrictos, un objeto no es más que una referencia *bendecida* con el nombre de la clase a la cual pertenece. Esta típicamente es una referencia a un hash, pero puede ser una referencia a cualquier tipo de datos.[9] Ahora bien, ¿En qué consiste la bendición?

A Perl le basta saber a qué clase pertenece un objeto pegándole una etiqueta con el nombre de la clase a la que pertenece. Para hacer esto usamos la función `bless`, a la cual le damos ya sea un sólo argumento (la referencia a ser bendecida) o dos (la referencia y el nombre de la clase). Si no se lo indicamos, tomará como nombre de la clase el espacio de nombres en el que es bendecida. Una vez bendecido, nuestra referencia deja de ser una simple referencia a un tipo de datos básico, para convertirse en un verdadero objeto, cosa que podemos verificar con la función `ref`:

```
$var = {llave=>valor};
print 'Lo que antes fue un ', ref($var);
bless $var, 'BenditoHash';
print ' es ahora un ', ref($var);
# Resultado: Lo que antes fue un HASH es ahora un BenditoHash
```

3.2.2. Construyendo el objeto

Cuando creamos una clase, tenemos que crear un *método constructor*, una función que será llamada con los argumentos requeridos por la clase y deseados

[7] Las diferentes tablas del sistema que nos sirven para guardar nuestras variables y funciones en su propio espacio — Para una explicación mejor y más completa, consulten el capítulo 10 de Programming Perl y los capítulos 3 y 6 de Advanced Perl Programming [15, 14]

[8] `@INC` es la lista de directorios donde Perl buscará las bibliotecas y los módulos que incluyamos con `use` o `require`, y normalmente incluye, además de lugares como `/usr/lib/perl5/`, `/usr/share/perl5/`, `/usr/local/lib/perl5/` y demás directorios del sistema al directorio actual (`.`), por lo que podemos trabajar sin problemas en nuestro directorio de desarrollo sin instalar nuestros módulos en su lugar definitivo. En caso de tener que agregar o eliminar alguna ruta de `@INC`, la mejor manera es por medio del pragma `use lib` / `no lib` — para más detalles, consulten la documentación [11].

[9] En el texto de Damian Conway [3] hay un muy buen ejemplo acerca de la implementación de un objeto utilizando una referencia a un arreglo, implementando además varias características de los objetos que no son normalmente utilizadas en Perl, como un verdadero encapsulamiento de datos. En este tutorial veremos sólamente la manera tradicional de manejo de objetos en Perl, a través de hashes.

por el usuario, y regresará una referencia bendecida — Un objeto.

La principal diferencia entre una función y un método es que el método recibe como primer argumento el objeto. El método que utilizaremos como constructor no puede, claro, recibir el objeto, pues este no ha sido aún creado — Lo que recibe como primer argumento es el nombre de la clase del objeto que creará.[10] El constructor típicamente recibe el nombre de `new`, pero esto es sólamente una convención. Varias clases implementan constructores con otros nombres por ser más lógicos para la labor que van a realizar (por ejemplo, la interfaz para el manejo de bases de datos, `DBI`, utiliza al método `connect` como constructor). De hecho, puede haber más de un constructor para una misma clase de objetos.

Con sólo esto, tenemos ya un objeto. Este objeto tendrá asociadas todas las funciones y atributos del objeto. Veamos el ejemplo:

```
package Objeto;
sub new {
    my $class = shift;
    my $obj = {}; # Referencia a un hash

    bless $obj, $class;
    return $obj;
}

package main;
my $obj = Objeto->new;
print 'Acabo de crear un ', ref($obj), "!\n";
__END__

Resultado de la ejecución:

Acabo de crear un Objeto!
```

¡Felicidades! Hemos creado nuestro primer (aunque tal vez un poco inútil) objeto.

3.2.3. Atributos de la clase y del objeto

Una importante característica de los objetos es que nos permiten guardar una colección de datos como parte del objeto, y que nos permiten guardar datos relativos a toda una clase de objetos. ¿Cómo? De la manera más simple posible, claro está.

Cada objeto tiene datos propios, los que lo diferencían de los demás objetos de su clase. Estos datos los conoceremos como *atributos de la instancia* o del objeto, y los representamos normalmente como los valores del hash referido

[10] Y, como veremos más adelante, *debemos* usarlo si no queremos dificultar el manejo de la herencia.

en el objeto[11]. En el caso de la clase `Gato` todos los atributos que definimos a excepción de `cantidad` son específicos a uno de los gatos y son, por tanto, atributos de la instancia.

La clase, además, puede guardar datos comunes a todos los objetos que ha creado. A ellos nos referiremos como *atributos de la clase*, y los representamos como variables normales de Perl en el espacio de nombres de la clase.

3.2.4. Métodos

Y la otra cosa importante que forma parte básica de la POO es la creación de métodos. Un método es simplemente una función que es llamada a través del objeto y que, por tanto, recibe como su primer argumento al objeto mismo.

A veces necesitamos hacer algo cada que un objeto deja de ser utilizado. Cuando tengamos esta necesidad, utilizaremos un *método destructor*. En nuestro caso, esto —afortunadamente— no es tan común como en otros lenguajes, dado que la memoria es manejada automáticamente por Perl y no hace falta liberar los recursos que ocupamos al crear o utilizar el objeto.

Un destructor se implementa creando un método llamado `DESTROY`. Al igual que con los demás métodos, éste recibirá al objeto como primer parámetro. A diferencia de los demás métodos, sin embargo, `DESTROY` nunca es llamado explícitamente, es Perl quien lo llama de manera automática al destruirse el objeto.

Vamos a un ejemplo más, volviendo a los gatos, ilustrando los últimos puntos mencionados. Cabe mencionar que este último ejemplo, si lo llamamos `Gato.pm`, es la implementación completa de la clase Gato que hemos venido comentando a lo largo de este texto.

```
package Gato;
# Atributos de clase
my $cantidad = 0;

# Constructor y destructor de clase
sub new {
    my $clase = shift;
    my $gato = {@_};

    # Atributos de instancia
    $gato->{lleno} ||= 0;
    $gato->{sueño} ||= 0;

    $cantidad++;
    bless $gato, $clase;

    return $gato;
```

[11] Nuevamente, recuerden que el lema de Perl es TIMTOWTDI, *There Is More Than One Way To Do It*. En este texto me limito a la manera más usual de hacerlo — Hay muchas más.

```
}

sub DESTROY {
    $cantidad--;
}

# Accesores y mutadores
sub lee_lleno { my $gato = shift; return $gato->{lleno}; }
sub lee_sueño { my $gato = shift; return $gato->{sueño}; }
sub lee_sexo { my $gato = shift; return $gato->{sexo}; }
sub lee_pelo { my $gato = shift; return $gato->{pelo}; }
sub lee_peso { my $gato = shift; return $gato->{peso}; }

sub pon_lleno {
    my $gato = shift;
    my $lleno = shift;
    # Valido que sea numérico y no demasiado lleno
    return undef unless ($lleno =~ /^\d+$/ and $lleno < 20);
    $gato->{lleno} = $lleno;
    return 1;
}

sub pon_sueño {
    my $gato = shift;
    my $sueño = shift;
    # Sólo valido que sea numérico
    return undef unless $sueño =~ /^\d+$/;
    $gato->{sueño} = $sueño;
    return 1;
}

sub pon_sexo {
    my $gato = shift;
    my $sexo = uc(shift);
    if defined $gato->{sexo} {
warn 'El sexo ya había sido definido';
return undef;
    }
    return undef unless $sexo =~ /[MF]/;
    $gato->{sexo} = $sexo;
    return 1;
}

sub pon_pelo { my $gato = shift; $gato->{pelo} = shift; }

sub pon_peso {
```

```
    my $gato = shift;
    $peso = shift;
    return undef unless $peso =~ /^\d+$/;
    $gato->{peso} = $peso;
    return 1;
}

# Métodos de instancia
sub come {
    my $gato = shift;
    my $cuanto_ingirio = shift;

    $gato->{lleno} += $cuanto_ingirio;
    $gato->vomita if ($gato->{lleno} > $gato->{peso} / 10);
    $gato->{sueño} += 2;

    return $gato->{lleno};
}

sub duerme {
    my $gato = shift;
    return 'No tengo sueño' if $gato->tiene_sueño < 1;
    $sueño = 0;

    return $sueño;
}

sub juega {
    my $gato = shift;
    return 'Estoy muy cansado' if $gato->tiene_sueño;
    return 'Tengo hambre' if $gato->tiene_hambre;
    $gato->{lleno} -= 1;
    $gato->{sueño} += 1;
    return 1;
}

sub vomita {
    my $gato = shift;
    $gato->{lleno} *= 2/3;
    return $gato->{lleno};
}

sub tiene_hambre {
    my $gato = shift;
    return ($gato->{lleno} < 5) ? 1 : 0;
}
```

```
sub tiene_sueño {
    my $gato = shift;
    return ($gato->{sueño} > 10) ? 1 : 0;
}

# Métodos de clase
sub cuantos {
    my $clase = ref($_[0]) || $_[0];
    return $cantidad;
}
```

Como pueden observar, todos los métodos de instancia comienzan diciendo `my $gato = shift;` Al llamar a un método, el primer argumento que le proporcionamos es el objeto mismo al cual nos estamos refiriendo. Los argumentos subsecuentes son manejados como en cualquier función.

En la clase `Gato` tenemos un sólo atributo de clase, la cantidad de gatos actualmente existentes. Este atributo está representado por la variable `$cantidad`, que, como habrán notado, está definida como léxica (con `my`). Esto hace que para consultar su contenido desde fuera de esta clase tengamos que hacerlo a través del método de clase `cuantos`. A veces querremos que el atributo sea directamente accesible desde fuera de la clase (como `$Gato::cantidad`). Para esto, la variable debe ser declarada con `our` en vez de `my`.

La primer línea del método de clase `cuantos` puede parecerles rara y, en este caso, innecesaria — La incluyo sólo para ilustrar cómo utilizamos los métodos de clase. Un método de clase no siempre recibirá un objeto como primer parámetro — Si es invocado directamente (por ejemplo, `$num = Gato::cuantos`) lo único que recibirá es el nombre de la clase. Si ya tuviéramos un objeto de la clase `Gato` creado y llamamos a este método a través de él (`$num = $tin_tan->cuantos`), recibirá un objeto. Con la línea `my $clase = ref($_[0]) || $_[0];` obtenemos ya sea el tipo de referencia que es el objeto o la clase a través de la cual invocamos al método.[12]

A veces tenemos clases simples, con unos cuantos atributos y sin mayor complicación. Otras veces, sin embargo, tenemos clases terriblemente complejas. Generar los accesores y mutadores es normalmente *talacha* — Trabajo muy simple, pero muy repetitivo. Si quieren ahorrar algo de trabajo al crear accesores y mutadores, puede interesarles revisar el módulo de CPAN `Class::Accessor` [12].

3.3. Yendo un poco más allá

Lo visto hasta ahora nos sirve para crear objetos simples. Sin embargo, la belleza de la POO no se aprecia tanto en sistemas simples como cuando reali-

[12] Y sí, siguiendo este razonamiento, el constructor `new` es nada menos que un método de clase.

zamos tareas mucho más complejas, que ya nos exigen implementar herencia, polimorfismo y... Lo mejor de nuestra imaginación :-)

3.3.1. Herencia

La herencia, como habíamos comentado, implementa una relación *es-un*, o *is-a*. Para representar que nuestra clase es subclase de otra, declaramos a cada una de sus superclases en su arreglo `@ISA`. Es importante recordar que `@ISA` *no* puede ser una variable de ámbito léxico (no puede haber sido declarada con `my`), pues Perl debe poder encontrarla por su nombre aún fuera de la clase — Podemos declararla con `our @ISA` en el módulo, o podemos llamarla por su nombre completo, `@Gato::ISA`. Si tenemos una implementación completa de `Animal`, basta con que declaremos `@Gato::ISA = qw(Animal);` para haber heredado todos sus métodos y atributos, y sólo nos queda revisar que especifiquemos/especialicemos los métodos únicos a un gato o los que un gato lleve a cabo de manera diferente.

Ahora bien, ¿por qué `@ISA` es un arreglo? Recordemos (subsección 1.3.2) que Perl permite la herencia múltiple — Y el órden en que mencione a las superclases importa. Si yo busco implementar la clase `Gato` como subclase de `Animal` y de `Entropizador`, tengo que pensar a qué superclase me es más importante heredar, y mencionarla primero, pues cuando Perl busca qué función llamar cuando el usuario solicita un método que no forma propiamente parte de la clase, hace una búsqueda por profundidad (*depth-first*). Supongamos que el árbol de herencia lo determinan los siguientes arreglos:

```
@Gato::ISA = qw(Animal Entropizador);
@Animal::ISA = qw(Ser_vivo);
@Entropizador::ISA = qw(Fuerza);
```

y solicitamos `$tin_tan->devora` (donde `devora` es un comportamiento heredado de una `Fueza`), Perl buscará la función actúa en `Gato`, `Animal`, `Ser_vivo`, `Entropizador` y finalmente lo encontrará en `Fuerza`. Sin embargo, si la función `devora` fuera provista por `Animal` (tal vez invocando a `come` con cierta brutalidad), sólamente la buscaría en `Gato` y `Animal`, y no seguiría buscando hacia Entropizador.

Muchas veces querremos únicamente modificar un poco el comportamiento de un método, no reimplementarlo por completo. Por ejemplo, normalmente después de dormir un gato se estira y sólo entonces podemos considerar que está despierto. Para hacer esto tenemos dos opciones: La primera, rígida y explícita, especificando la clase:

```
sub duerme { my $gato = shift; $gato->Animal::duerme; $gato->estira;
return 1; }
```

La segunda, más genérica y (siempre que sea posible) más recomendable:

```
sub duerme { my $gato = shift; $gato->SUPER::duerme; $gato->estira;
return 1;}
```

Esta segunda manera nos da mayor independencia. Y si el método `duerme` se moviera de Animal a alguna otra clase superior, esta segunda versión seguiría funcionando sin modificación. La sintaxis SUPER se refiere a cualquiera de

nuestras superclases. Claro está, sólo tiene significado dentro de un método de esta clase.

Una última anotación: El listar dentro de nuestro `@ISA` a una clase no significa que estemos leyendo su código del disco. Cuando estemos heredando de una clase, debemos especificar a Perl también (con `use Clase;`) que cargue el archivo donde está el código de la clase.

3.3.2. Polimorfismo

Cuesta trabajo definir la herencia sin haber incluído en la definición al polimorfismo y, de hecho, yo no lo intenté siquiera. El polimorfismo simplemente significa que, cuando una clase hereda de la otra, no es una herencia rígida, sino que me permite *especializar* a los métodos — Darle a un mismo método diferentes comportamientos ante diferentes tipos de objeto.

El polimorfismo nos permite programar de una manera mucho más genérica. Si estuviéramos programando un `Veterinario`, tenemos que escribir el código únicamente una vez, y gracias al polimorfismo podríamos emplearlo para los diferentes animales que lleguen a consulta.

El polimorfismo no es tan impresionante ni novedoso en Perl como en lenguajes fuertemente tipificados, como C++ o Java, en los que cada variable tiene un tipo de datos —una clase— que puede alojar, y nada más cabe en ella. En estos lenguajes, los métodos de `Veterinario` aceptarían objetos de tipo `Animal` y, como todo `Gato` y `Caballo` son `Animal`es, el programador del `Veterinario` no tiene que volverse loco contemplando todos los posibles escenarios de uso. El `Veterinario` incluso podría llevar a cabo una revisión de rutina a un `Ornitorrinco` que casualmente estuviera en el país.

3.3.3. Sobrecarga de operadores

Si queremos implementar sobrecarga de operadores, lo podemos hacer utilizando el módulo `overload` [13], parte de la distribución estándar de Perl. Supongamos que tenemos una clase que implementa números del 0 al 255 usando un byte. Decidimos implementar los métodos `add`, `sub`, `mult` y `div`, que son todas las operaciones que manejaremos. Por ahora, ignoren la curiosa línea que inicia con `use overload`.

```
package Byte;

use overload '+' => \&add, '-' => \&sub, '*' => \&mult, '/' => \&div,
    '""' => \&value, '0+' => \&value;

sub new {
    my ($class, $val) = @_;
    my $byte = { -val => pack('c',$val) };
    bless $byte, $class;
    return $byte;
```

```
}
sub value {
    my $byte = shift;
    return unpack('c',$byte->{-val})
}
sub add {
    my ($byte, $num) = @_;
    return Byte->new($byte->value + $num);
}
sub sub {
    my ($byte, $num) = @_;
    return Byte->new($byte->value - $num);
}
sub mult {
    my ($byte, $num) = @_;
    return Byte->new($byte->value * $num);
}
sub div {
    my ($byte, $num) = @_;
    return Byte->new($byte->value / $num);
}
1;
```

Perfecto. Sin embargo, manejarlo puede ser bastante incómodo. Un simple ejemplo:

```
$a = Byte->new(50);
$b = Byte->new(15);
$c = $a->add($b->value);
print $c->value, ' es un ', ref($c);
```

Todo este ilegible proceso es necesario para sumar 50 y 15 e imprimir el valor. Sin embargo, gracias al uso de `overload`, lo siguiente es perfectamente equivalente:

```
$a = Byte->new(50);
$b = Byte->new(15);
$c = $a + $b;
print $c, ' es un ', ref($c);
```

Y mágicamente, nuestro `Byte` se comporta ya como un tipo de datos nativo de Perl (claro, únicamente con los operadores que le definimos. Si le pedimos una operación que no hemos definido como calcular el módulo (`%`) el programa abortará quejándose `Operation '%': no method found`).

Nuestra última instrucción imprime el valor del nuevo byte, 65, sin quitar el hecho de que `ref` nos revele que debajo sigue existiendo un objeto Byte, silenciosamente creado y manejado por Perl.

3.4. Por fin, el fin

Y con esto queda, a mi entender, cubierta la introducción a los objetos de Perl. ¿Qué sigue? Pues... Echar código, que es la única manera de dominar el tema.

Referencias

[1] Grupo de Usuarios de Linux del Estado de Veracruz (GULEV), Liga Web `http://www.gulev.org.mx`

[2] Procesador de documentos LyX, Liga Web `http://www.lyx.org/`

[3] What is Object Oriented Perl? — Damian Conway PDF: `http://www.csse.monash.edu.au/~damian/papers/PDF/cyberdigest.pdf`

[4] Object Oriented Perl — Damian Conway, ed. Manning Publications 1999, ISBN 1-884777-79-1

[5] Beginning Perl (capítulo 11) — Simon Cozens, Peter Wainwright, ed. Wrox press 2000, ASIN 1861003145 Libro en línea `http://learn.perl.org/library/beginning_perl/`

[6] Tratado sobre los Ángeles, *Summa Theologica*, Santo Tomás de Aquino

[7] perldoc perlform — Perl formats, Liga Web `http://www.perldoc.com/perl5.8.0/pod/perlform.html`

[8] Seven uses of Local — Mark Jason Dominus, The Perl Journal 1999, Liga Web `http://perl.plover.com/local.html`

[9] perldoc strict — Perl pragma to restrict unsafe constructs, Liga Web `http://www.perldoc.com/perl5.8.0/pod/strict.html`

[10] perldoc perlref — Perl references and nested data structures, Liga Web `http://www.perldoc.com/perl5.8.0/pod/perlref.html`

[11] perldoc lib — Manipulate @INC at compile time, Liga Web `http://www.perldoc.com/perl5.8.0/pod/perlref.html`

[12] Módulo `Class::Accessor` del CPAN, Michael G. Schwern, Liga Web `http://www.cpan.org/modules/by-module/Class`

[13] perldoc overload — Package for overloading Perl operations, ligaurl `http://www.perl.com/doc/manual/html/lib/overload.html`

[14] Programming Perl 3^a edición (capítulos 8, 10, 11, 12) — Larry Wall, Tom Christiansen, Jon Orwant, ed. O'Reilly 2000, ISBN 0-596-00027-8

[15] Advanced Perl Programming (capítulos 1, 3, 4, 6, 7, 8) Sriram Srinivasan, ed. O'Reilly 1997, ISBN 1-56592-220-4

[16] Perl Cookbook (capítulos 11, 12, 13) Tom Christiansen, Nathan Torkington, ed. O'Reilly 1998, ISBN 1-56592-243-3

[17] perldoc perlboot — Beginner's Object Oriented Tutorial, Liga Web `http://www.perldoc.com/perl5.8.0/pod/perlboot.html`

[18] perldoc perltoot — Tom's object-oriented tutorial for perl, Liga Web `http://www.perldoc.com/perl5.8.0/pod/perltoot.html`

[19] perldoc perltooc — Tom's OO Tutorial for Class Data in Perl,Liga Web `http://www.perldoc.com/perl5.8.0/pod/perltooc.html`