Object-Oriented Perl — A Quick Review

by

Avinash Kak
kak@purdue.edu

Last Modified: June 15, 2006

- This review was created specially for the students taking ECE 495F, **Introduction to Computer Security**, to help them with writing Perl scripts for solving homework problems.

- This review was excerpted from the forthcoming book "**Scripting with Objects: A Comparative Presentation of Perl and Python**" by Avinash Kak

- Avinash Kak is also the author of "**Programming with Objects: A Comparative Presentation of Object-Oriented Programming with C++ and Java**", published by John-Wiley & Sons, New York, 2003. This book presents a new approach to the combined learning of two large object-oriented languages, C++ and Java. It is being used as a text in a number of educational programs around the world. This book has also been translated into Chinese. For further information, visit `http://programming-with-objects.com`.

The Main Concepts

The following fundamental notions of object-oriented programming in general apply to object-oriented scripting also:

- **Class**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

What is a Class

- At a high level of conceptualization, a class can be thought of as a category. We may think of "Cat" as a class.

- A specific cat would then be an **instance** of this class.

- For the purpose of writing code, a class is a data structure with **attributes**.

- An instance constructed from a class will have specific values for the attributes.

- To endow instances with behaviors, a class can be provided with **methods**.

- A **method** is a function that you **invoke on** an instance of the class or the class itself.

- A method that is invoked on an instance is sometimes called an **instance method**.

- You can also invoke a method directly on a class, in which case it is called a **class method** or a **static method**.

- Attributes that take values on a per-instance basis are sometimes referred to as **instance attributes** or **instance variables**.

- You can also have attributes that take on values on a per-class basis. Such attributes are called **class attributes** or **static attributes** or **class variables**.

Encapsulation

- Hiding, or controlling the access to, the implementation-related attributes and the methods of a class is called **encapsulation**.

- With appropriate data encapsulation, a class will present a well-defined **public interface** for its **clients**, the users of the class.

- A client should only access those data attributes and invoke those methods that are in the **public interface**.

Inheritance and Polymorphism

- **Inheritance** in object-oriented code allows a subclass to inherit some or all of the attributes and methods of its superclass(es).

- **Polymorphism** basically means that a given category of objects can exhibit multiple identities at the same time, in the sense that a `Cat` object is not only of type `Cat`, but also of types `FourLegged` and `Animal`, *all at the same time.*

- Therefore, if we were to make an array like

```
@animals = (kitty, fido, tabby, quacker, spot);
```

of cats, dogs, and a duck — *instances made from different classes in some* `Animal` *hierarchy* — and if we were to invoke a method named `calculateIQ()` on this list of animals in the following fashion

```
foreach my $item (@animals) {
    $item.calculateIQ();
}
```

polymorphism would cause the correct implementation code for `calculateIQ()` to be automatically invoked for each of the animals.

More on the Previous Example on Polymorphism

- In many object-oriented languages, a method such as `calculateIQ()` would need to be declared for the root class `Animal` for the control loop shown on the previous page to work properly.

- All of the public methods and attributes defined for the root class would constitute the public interface of the class hierarchy and each class in the hierarchy would be free to provide its own implementation for the methods declared in the root class.

- The sort of loop we showed on the previous page would constitute manipulating the instances made from the classes in the hierarchy through the public interface defined by the root class.

- Polymorphism, in a nutshell, allows us to manipulate instances constructed from the different classes of a hierarchy through a common interface defined for a root class.

Defining a Class in Perl

- We need to pull together three concepts in order to understand the notion of a class in Perl:
  - packages
  - references
  - the notion of **blessing an object into a package**

Using a Perl Package as a Class

- A class in Perl is a package that encloses the methods that characterize the behavior of the class and the instances constructed from the class.

- Since, strictly speaking, a package is merely a namespace, it does **not** allow us to directly incorporate the attributes that can be used on a **per-instance** basis in the same sense that a C++ or a Java class does. (Although, as we will see later, a package does allow us to use attributes on a per-class basis, meaning like the static data members in C++ and Java.)

- This limitation, however, does not prevent us from defining an instance with state.

- Perl creates stateful instances through the expedient of packaging the state variables (meaning the per-instance data attributes) inside a standard data structure like a hash and having an instance constructor return such a hash as an instance of a class.

- It is important to realize that just as much as Perl can use a hash for a class instance, it can also use a scalar or an array.

- The convenience of a hash is obvious — it gives us named placeholders for the instance variables, just like the data members in a C++ class or a Java class.

- So constructing an instance of a class is no different from constructing a hash, or an array, or just a scalar.

- But hashes, arrays, and scalars in Perl are free-standing objects in Perl, meaning they don't ordinarily belong to any particular package.

- It is indeed true that, in general, a variable inside a package would need to be accessed with its package qualified name. But if the variable is holding a reference to, say, a hash, that hash itself has no particular package association.

- So how does Perl establish the needed association between a data object that is to serve as an instance of a class and the class itself (meaning the package that will be used as a class)?

- Said another way, how does Perl acquire the notion that a data object is of a certain type?

- The type labeling is needed if the behavior of the object must

correspond to what is specified for that class through its methods.

- In Perl, the type association between a data object and a package that is to serve as a class is established through the mechanism of **blessing**.

- When an object is blessed into a package, the object becomes tagged as belonging to the package.

- Subsequently, the object can be considered to be of the type that is the name of the package.

Blessing an Object into a Class

- Notice the type of the reference held by `$ref` before and after it is blessed:

```
my $ref = { name => "Trillian", age => 35 };
print ref( $ref ), "\n";            # HASH
bless $ref, Person;
print ref( $ref ), "\n";            # Person
```

  By the way, the above call will also create a class named `Person` by the autovivification feature of Perl.

- After an object has been blessed in the manner shown above, a method invoked via the arrow operator on a reference to the object will try to call on the subroutine of that name in the package into which the object was blessed.

```
package Person;

    sub get_name {
        my $self = shift;
        $self->{name};
    }

package main;
    my $ref = { name => "Trillian", age => 35 };
    bless $ref, Person;
    print $ref->get_name(), "\n";     # Trillian
```

- Note that the subroutines defined for packages that are meant to serve as classes are written a bit differently compared to regular subroutines.

- For example, the subroutine `get_name()` expects one argument that must be a reference to a blessed object.

- When this subroutine is invoked without any explicit arguments on a reference to a blessed object, the subroutine invocation is translated by Perl into the following function call:

```
Person::get_name( $ref )
```

  which agrees with how the subroutine expects itself to be called.

Can Any Reference to Any Sort of an Object be Blessed?

- Any reference whatsoever in Perl can be blessed. Here we are blessing an array:

```
package StringJoiner;

sub wordjoiner {
    my $self = shift;
    join "", @$self;
}

package main;

my $ref = ['hello', 'jello', 'mello', 'yello'];
bless $ref, StringJoiner;
print $ref->wordjoiner(), "\n";   # hellojellomelloyello
```

- Note again that because `$ref` is a reference blessed into the `StringJoiner` package, Perl translates the arrow operator based method call `$ref->wordjoinder()` into the following call

```
StringJoiner::wordjoiner( $ref )
```

Providing a Class with a Constructor

- A constructor in Perl must do the following:
  - select a storage mechanism for the instance variables of the class;
  - obtain a reference to the data object created (which will serve as a class instance) for the values provided for the instance variables; and
  - bless the data object into the class and return the blessed reference to the object.
- There is also the issue of what to call the constructor.
- While there are constraints on how a constructor is named in Python and in mainstream OO languages like C++ and Java, that is not the case with Perl.
- In Perl, if a class has a single constructor, it will typically be named `new`. But a class is allowed to have any number of constructors, a feature that Perl has in common with C++ and Java.

- The constructor shown below uses a hash as a storage mechanism for the instance objects. That will usually be the case for most classes in Perl.

```
package Person;

sub new {
    my ($class, $name, $age) = @_;
    bless {
        _name => $name,
        _age  => $age,
    }, $class;
}
```

- The constructor shown above is typically invoked with the following arrow-operator based syntax:

```
my $person = Person->new( "Zaphod", 114 );
```

  Subsequently, the variable **`$person`** will hold a reference to an instance of class `Person`.

- Note that the arrow operator is now being used to invoke a method on the class itself, as opposed to on a reference to an instance object that we showed earlier. Perl translates this call into

```
my $person = Person::new( "Person", "Zahpod", 114 );
```

- If we so wanted, we could have used the above syntax directly for constructing an instance of the `Person` class.

- But that is not a recommended style for Perl OO because of its ramifications that will become apparent when we talk about constructing instances of subclasses of a class.

Data Hiding and Data Access Issues

- In the Person class, the names we used for the attributes started with an underscore.

- This is just a convention in Perl OO for denoting those names that are internal to a class.

- Ideally, it should be nobody's business how the various attributes are represented inside a class, not even how they are named.

- Unlike in C++ and Java, there is no way to enforce such privacy aspects of how data is stored in a class in Perl.

- But since it is nonetheless important to honor such privacy, in Perl we resort to conventions.

- The convention regarding data hiding states that the names used for the instance and the class variables begin with an underscore and that these attributes of a class be accessible only through the methods designated specifically for that purpose, as shown below for a class with two instance variables `_name` and `_age`:

```
package Person;

sub new {
    my ($class, $name, $age) = @_;
    bless {
        _name => $name,
        _age  => $age,
    }, $class;
}

sub name { $_[0]->{_name} }

sub age  {
    my ($self, $age) = @_;
    $age ? $self->{_age} = $age : $self->{_age};
}
```

Packaging a Class into a Module

- A package in Perl is merely a namespace and it is indeed possible to have multiple packages in the same script file.

- But when it comes to using a package as a class in the object-oriented sense, a common practice is to have a single package — and therefore a single class — in a file, thus creating a module file for a class.

- The suffix for such a file, as for all modules, is '`.pm`'.

```
package Person;

### Person.pm

use strict;

sub new {
    my ($class, $name, $age) = @_;
    bless {
        _name => $name,
        _age  => $age,
    }, $class;
}

sub get_name { $_[0]->{_name}; }
```

```
sub get_age { $_[0]->{_age}; }

sub set_age {
    my ($self, $age) = @_;
    $self->{_age} = $age;
}

1
```

- Next we show how this class can be used in a test script:

```
#!/usr/bin/perl -w

### TestPerson.pl

use strict;

use Person;

my ($person, $name, $age);
$person = Person->new( "Zahpod", 114 );
$name = $person->get_name;
$age = $person->get_age;
print "name: $name   age: $age\n";  # name: Zaphod   age: 114

$person->set_age(214);
$age = $person->get_age;
print "name: $name   age: $age\n";  # name: Zaphod   age: 214
```

Constructors with Named Parameters

- When a function takes a large number of arguments, it can be difficult to remember the position of each argument in the argument list of a function call.

- Fortunately, Perl scripts can take advantage of the built-in hash data structure so that functions can be called with named arguments.

- In addition to the convenience provided by attaching a name with an argument, the name and argument pairs can be specified in any order.

- The same can be done for a constructor. If a constructor takes a large number of arguments, it is more convenient to be able to supply them in an arbitrary positional order and with associated names.

- The constructor of the following class expects named arguments:

```
package Employee;

### Employee.pm
```

```
use strict;

sub new {
    my ($class, %args) = @_;
    bless {
        _name           =>   $args{ name },
        _age            =>   $args{ age },
        _gender         =>   $args{ gender },
        _title          =>   $args{ title },
        _dept           =>   $args{ dept },
        _grade          =>   $args{ grade },
    }, $class;
}

sub get_name { $_[0]->{_name} }
sub get_age { $_[0]->{_age} }
sub get_gender { $_[0]->{_gender} }
sub get_title { $_[0]->{_title} }
sub get_dept { $_[0]->{_dept} }
sub get_grade { $_[0]->{_grade} }

sub set_age { $_[0]->{_age} = $_[1] }
sub set_title { $_[0]->{_title} = $_[1] }
sub set_dept { $_[0]->{_dept} = $_[1] }
sub set_grade { $_[0]->{_grade} = $_[1] }

1
```

- The constructor of the above class can be called with the following more convenient syntax:

```
my $emp  = Employee->new( name   => "Poly",
                          title  => "boss",
                          dept   => "fleet",
                          age    => 28,
                          gender => "female",
                          grade  => "junior");
```

Default Values for Instance Variables

- If desired, it is possible to provide a constructor with default values for one or more of the instance variables named in the body of the constructor.

- When the constructor is meant to be called with the arguments in a specific positional order, the default values can only be specified for what would otherwise be the trailing arguments in a normal constructor call.

- With a constructor that expects to be called with named arguments, any argument left unspecified can be taken care of by its default value.

- Shown below is a class, `Flower`, whose constructor specifies default values (or, more precisely speaking, actions) for each of the three attributes. Using the '`||`' operator, the default values are provided as disjunctions to the parameters of the constructor.

```
package Flower;

### Flower.pm

use strict;
```

```
use Carp;

sub new {
    my ($class, $name, $season, $frag) = @_;
    bless {
        _name        =>   $name   || croak("name required"),
        _season      =>   $season || _ask_for_season($name),
        _fragrance   =>   $frag   || 'unknown',
    }, $class;
}

sub get_name { $_[0]->{_name}; }
sub get_season { $_[0]->{_season}; }
sub get_fragrance { $_[0]->{_fragrance}; }

sub set_season { $_[0]->{season} = $_[1] }
sub set_fragrance { $_[0]->{fragrance} = $_[1] }

sub _ask_for_season {
    my $flower = shift;
    print STDOUT "enter the season for $flower: ";
    chomp( my $response = <> );
    $response;
}

1
```

- When a constructor is defined to take arguments through "(name, value)" pairs, any arbitrary attribute can be provided with a default at the instance construction time.

Hiding Free-Standing Functions

- The `Flower` class suffers from one limitation: It does not make it difficult for a programmer to try to use the subroutine `_ask_for_season()` directly even though that subroutine is meant for just the internal use by the class.

- A client of the `Flower` class could, for example, make the following invocation:

```
my $flower = Flower->new( "Rose" );
$flower->_ask_for_season();
```

  While the result of this external invocation of `_ask_for_season()` would produce a meaningless result in this case, since the instance object, as opposed to the name of a flower, would be passed as the argument to the subroutine, in general anything could happen, including the injection of a difficult to locate bug in a large script.

- Such problems can be fixed by making it impossible to access all subroutines that are intended solely for internal purposes in a class.

```
package Flower;
```

```
### Flower.pm
use strict;
use Carp;

my $ask_for_season = sub {
    my $flower = shift;
    print STDOUT "enter the season for $flower: ";
    chomp( my $response = <> );
    $response;
};

sub new {
    my ($class, $name, $season, $frag) = @_;
    bless {
        _name        =>   $name   || croak("name required"),
        _season      =>   $season || $ask_for_season->($name),
        _fragrance   =>   $frag   || 'unknown',
    }, $class;
}

sub get_name { $_[0]->{_name}; }
sub get_season { $_[0]->{_season}; }
sub get_fragrance { $_[0]->{_fragrance}; }

sub set_season { $_[0]->{season} = $_[1] }
sub set_fragrance { $_[0]->{fragrance} = $_[1] }
1
```

Object Destruction

- Perl comes with an automatic garbage collector for reclaiming memory occupied by unreferenced objects.

- The garbage collector is invoked by the system automatically through the mechanism of reference counting that is associated with the objects.

- In the same spirit as a destructor in C++ and the `finalize()` method in Java, Perl allows a programmer to define a special method named `DESTROY()` that is called automatically for cleanup just before the system reclaims the memory occupied by the object.

- This can be important in situations where an object contains open filehandles, sockets, pipes, database connections, and other system resources. The code to free up these resources can be placed in the `DESTROY()` method.

- In addition to being invoked when the reference count for an object goes down to zero, `DESTROY()` would also be called if the process or the thread in which the Perl interpreter is running is shutting down.

Class Variables and Methods

- Except for the constructors and a few other functions embedded in class definitions, the subroutines shown in the previous class definitions have mostly been those that are meant to be invoked on a per instance basis. A constructor is obviously intended to be invoked directly on a class.

- Methods that are meant to be invoked directly on a class are commonly referred to as either *class methods* or as *static methods*.

- We will now show how one sets up a static method in Perl (apart from the constructor). We will also show how you can make sure that a class method is not invoked on an instance of the class.

- Just like class methods, we can also have *class variables* or *class attributes* or `static attributes`. These variables are global with respect to the class, meaning global with respect to all the instances made from that class.

- Shown next is a `Robot` class that allows us to assign a unique serial number to each instance made from the class. The class has

been provided with a class-based storage (as opposed to instance-based storage) for keeping track of the serial numbers already assigned so that the next `Robot` instance would get the next serial number.

- The class also has been provided with a class method for returning the total number of robots already made.

```
#!/usr/bin/perl -w
use strict;

### ClassData.pl

package Robot;

my $_robot_serial_num = 0;
my $_next_serial =   sub { ++$_robot_serial_num };
my $_total_num   =   sub { $_robot_serial_num };

# Constructor:
sub new {
    my ( $class, $owner ) = @_;
    bless {
        _owner => $owner,
        _serial_number => $_next_serial->(),
    }, $class;
}

# This instance method can work both as a 'set'
# and a 'get' method:
sub owner {
    my $self = shift;
```

```
    @_ ? $self->{_owner} = shift
       : $self->{_owner};
}

# This is an instance method:
sub get_serial_num {
    my $self = shift;
    $self->{_serial_number};
}

# This is a class method:
sub how_many_robots {
    my $class = shift;
    die "illegal invocation of a static method"
        unless $class eq 'Robot';
    $_total_num->();
}

#-----------------------------------------------------------------
#                           Test Code
#-----------------------------------------------------------------
package main;

my $bot = Robot->new( "Zaphod" );
my $name = $bot->owner();
my $num = $bot->get_serial_num();
print "robot owner: $name   serial number: $num\n";
                      # robot owner: Zaphod   serial number: 1

$bot = Robot->new( "Trillian" );
$name = $bot->owner();
$num = $bot->get_serial_num();
print "robot owner: $name   serial number: $num\n";
                      # robot owner: Trillian   serial number: 2
```

```
$bot = Robot->new( "Betelgeuse" );
$name = $bot->owner();
$num = $bot->get_serial_num();
print "robot owner: $name   serial number: $num\n";
                        # robot owner: Betelgeuse   serial number: 3

my $total_production = Robot->how_many_robots();
print "Total number of robots made: $total_production\n";
                        # Total number of robots made: 3

#my $x = Robot::how_many_robots();    #ERROR
```

Inheritance and Polymorphism in Mainstream OO

- Inheritance in mainstream OO languages such as C++ and Java means that a derived class inherits the attributes and the methods of all its parent classes.

- What the word *inherits* means in the above sentence is tighter than what would be suggested by, say, a child class just acquiring the attributes and the methods of its parent class.

- In the mainstream OO languages, the memory allocated to an instance of a child class contains slots for all the instance variables in all the parent classes.

- So an instance of a child class has built into it "sub-instances" of the parent classes.

- It is for this reason that a constructor for a child class must explicitly or implicitly call the constructor of its parent classes before it does anything else. The calls to the constructors of the parent classes are needed for the initialization of that portion of the memory of a child class instance that is meant to hold the parent-class slices.

Inheritance and Polymorphism in Perl

- Inheritance in Perl (and also Python) works very differently compared to how it works in mainstream OO languages such as C++ and Java.

- Perhaps the biggest difference is caused by the fact that when memory is allocated for a subclass instance in Perl (and in Python also), it does not contain slots for the base class attributes.

- In other words, a subclass instance in Perl is a completely separate data object and it does not contain "slices" for the data objects that could be formed from the parent classes.

- So there is no direct coupling between subclass instances and parent class instances. Therefore, it is not necessary for a subclass constructor to always invoke the constructors of the base classes in order to initialize the memory slots reserved for the base class slices inside subclass instances.

- Inheritance in Perl (and also Python) means only that if a method invoked on a subclass instance is not found within the subclass definition itself, it will be *searched for* in the parent classes.

- This manner of search for a method in the parent classes (if a definition cannot be found directly in the class that corresponds to the instance on which the method is invoked) automatically allows class instances to behave polymorphically, *albeit with a subtle twist vis-a-vis how polymorphism works in mainstream OO languages.*

- The polymorphism that results from Perl's search-based approach to inheritance differs in a subtle way from polymorphism in mainstream OO languages. Whereas a child class in mainstream OO languages is *equally polymorphic* — if one could use that characterization — with respect to all its parent classes, polymorphic behavior in Perl is weighted more toward the parent classes that appear earlier in the recursive depth-first left-to-right search order through the parent classes of a child class.

- Perl (and Python also) shares with the mainstream OO languages many of the other benefits derived from inheritance. As in the mainstream OO languages, class hierarchies in Perl can be used for incremental development of code. Perl (and Python also) allow for the definition of abstract classes that can be used to declare interfaces to which users of a class hierarchy can program to.

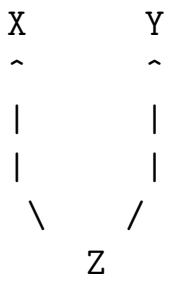
The ISA Array for Specifying the Parents of a Class

- The `ISA` array is fundamental to how inheritance works in Perl. Each derived class is provided with a list of its parent classes through this array.

```
#!/usr/bin/perl -w
use strict;

###  InheritanceBasic.pl

package X;
    sub new { bless {}, $_[0] }
    sub foo { print "X's foo called\n" }

package Y;
    sub new { bless {}, $_[0] }
    sub bar { print "Y's bar called\n" }

package Z;
    @Z::ISA = qw( X Y );
    sub new { bless {}, $_[0] }

#----------------------  end of class definitions --------------------

package main;

print join ' ', keys %Z::, "\n";   # ISA new
```

```
my $zobj = Z->new();
$zobj->foo();                          # X's foo called
print join ' ', keys %Z::, "\n";   # ISA foo new

$zobj->bar();                          # Y's bar called
print join ' ', keys %Z::, "\n";   # bar ISA foo new
print join ' ', values %Z::, "\n";
                  #  *Z::bar *Z::ISA *Z::foo *Z::new
```

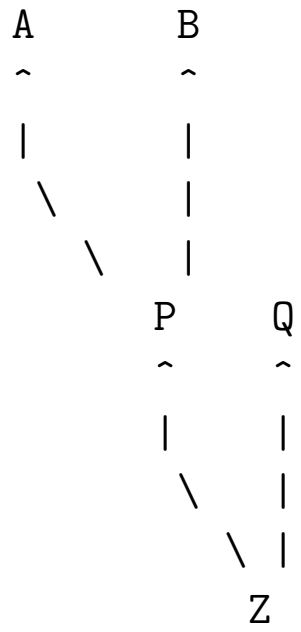
- By virtue of the `ISA` declaration for class `Z`, the three classes together, `X`, `Y`, and `Z`, constitute the following inheritance hierarchy:



- So when we invoke a method of an instance of class `Z`, if that method's definition does not exist in class `Z` itself, the method would be searched for in the parent classes in the `ISA` array defined for class `Z`. In other words, such a function call would be dispatched up the inheritance tree.

- When not found in the class itself, the search for a method definition in the inheritance tree takes place through a recursive left-

to-right depth-first fashion. Let's say the inheritance tree looks something like:

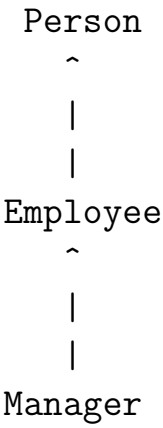


When a method invoked on an instance of `Z` is not found within the definition of `Z` itself, the method name would be looked for in the namespace of class `P`. If not found there either, the search would move into the namespace of class `A`. From `A` it will go to `B`, before moving into `P` sibling `Q`.

- This search for a method takes place only once. Subsequently, a reference to that method is cached in the namespace of the class itself.

An Example of Class Derivation in Perl

We will now illustrate class derivation with the three class hierarchy shown below:




```
#!/usr/bin/perl -w
use strict;

### PersonHierarchy.pl

#-------------------- class Person ------------------------
package Person;

sub new {
    my ( $class, $name, $age ) = @_;
    my $instance = {
        _name => $name,
        _age => $age,
    };
    bless $instance, $class;
```

```
}
sub get_name {
    my $self = shift;
    $self->{_name};
}
sub get_age {
    my $self = shift;
    $self->{_age};
}

#--------------------- class Employee -------------------
package Employee;

@Employee::ISA = ('Person');

sub new {
    my ( $class, $name, $age, $position ) = @_;
    my $instance = Person->new( $name, $age );
    $instance->{_position} = $position;
    bless $instance, $class;
}
sub get_position {
    my $self = shift;
    $self->{_position};
}
my %_promotion_table = (
    shop_floor => "staff",
    staff => "management",
    astt_manager => "manager",
    manager => "executive",
);
sub promote {
    my $self = shift;
    $self->{_position} =
```

```
                    $_promotion_table{ $self->{_position} };
}

#--------------------  class Manager  ----------------------
package Manager;

@Manager::ISA = ("Employee");

sub new {
    my ( $class, $name, $age, $position, $department ) = @_;
    my $instance = Employee->new( $name, $age, $position );
    $instance->{_department} = $department;
    bless $instance, $class;
}
sub get_department {
    my $self = shift;
    $self->{_department};
}
```

The script that follows is for testing the `Person` hierarchy:

```
#!/usr/bin/perl -w
use strict;

### TestPersonHierarchy.pl

require "PersonHierarchy.pl";

my ($name, $age, $position, $dept);
```

```
my $emp = Employee->new( "Zaphod", 84, "shop_floor" );
$name = $emp->get_name();
$age = $emp->get_age();
$position = $emp->get_position();
print "$name   $age   $position\n";
                        # Zaphod   84   shop_floor

$emp->promote();
$position = $emp->get_position();
print "$name   $age   $position\n";
                        # Zaphod   84   staff

my $man = Manager->new( "Trillian", 42, "astt_manager", "sales" );
$name = $man->get_name();
$age = $man->get_age();
$position = $man->get_position();
$dept = $man->get_department();
print "$name   $age   $position   $dept\n";
                        # Trillian   42   astt_manager   sales
$man->promote();
$position = $man->get_position();
print "$name   $age   $position   $dept\n";
                        # Trillian   42   manager   sales
```

Demonstration of Polymorphism for the Person Class Hierarchy

- By placing `Person` in the `ISA` array of `Employee` and `Employee` in the `ISA` array of `Manager`, an `Employee` instance inherits the methods of the `Person` class and a `Manager` instance inherits the methods of both the `Person` class and the `Employee` class. This would allow an `Employee` to act like a `Person`. This would also allow a `Manager` to act like an `Employee` and like a `Person`. That is polymorphism in its most basic form.

- For strongly typed OO languages like C++ and Java, a particular consequence of polymorphism is the following property: *A derived class type can be substituted wherever a base class type is expected.*

- So, in C++ and Java, if you write a function that needs a base class argument, by virtue of polymorphism you could call this function with a derived class object for the argument. In other words, suppose we write a function in Java

```
void foo( Person p );
```

  we can invoke the function in the following manner:

```
Manager man = new Manager( ..... );
foo( man );
```

- This manifestation of polymorphism in Perl is displayed by the script shown next, `Polymorph.pl` . Note that the following subroutine in the script is meant to take a `Person` argument:

```
sub foo {               #  expects a Person argument
    my ($arg) = @_;
    my $nam = $arg->get_name();
    print ‘‘subroutine foo reporting: $nam\n’’;
}
```

- The subroutine expects a Person argument because the `get_name()` function that is invoked in the body of the subroutine is defined specifically for the `Person` class. Nonetheless, when the function `foo()` is invoked with an `Employee` argument in line (H) and with a `Manager` argument in line (I), the program behaves as a typical OO program — it exhibits polymorphism.

```
#!/usr/bin/perl -w
use strict;

### Polymorph.pl

require "PersonHierarchy.pl";

sub foo {                          #  expects a Person argument
    my ($arg) = @_;
    my $nam = $arg->get_name();
    print "subroutine foo reporting: $nam\n";
}

my $per = Person->new( "Zaphod", 84 );

my $emp = Employee->new( "Orpheus", 84, "shop_floor" );

my $man = Manager->new( "Trillion", 42, "astt_manager", "sales" );

#foo invoked on a Person:
foo( $per );                           #Zahpod

#foo invoked on an Employee:
foo( $emp );                           #Orpheus

#foo invoked on a Manager:
foo( $man );                           #Trillian
```

How can a Derived Class Method call on a Base Class Method?

- We will now show how a derived-class method can call a base-class method of the same name for doing part of the work and how this is accomplished with the help of the keyword **SUPER**.

- Consider the following script with a two-class hierarchy:

```
Employee
   ^
   |
   |
Manager
```

- Both the `Employee` class and the `Manager` subclass are provided with the following two methods:

```
promote()
print()
```

  but we want the implementations of these methods for the subclass `Manager` to use the implementations in the base class.

- In other words, we want the `print()` method of `Manager` to call on the `print()` method of `Employee` for doing part of the

work. Along the same lines, we want the `promote()` method of `Manager` to call on the `promote()` method of `Employee`.

- Let's focus on the `print` method. Whereas its implementation for the base class is as you'd expect, note how the derived class implementation calls on the base class implementation by using the syntax:

```
$instance->SUPER::print();
```

- The keyword `SUPER` causes Perl to search for `print()` in the direct and indirect superclasses of `Manager`. This search proceeds in exactly the same fashion as for any regular method invocation, except that it starts with the direct superclasses.

- It is also possible to ask Perl to confine its search to a particular superclass (and all the superclasses of that class):

```
$instance->Employee::promote();
```

```
#!/usr/bin/perl -w
use strict;

###  Super.pl

#--------------------  class Employee  ----------------------
package Employee;

sub new {
    my ( $class, $name, $position ) = @_;
    my $instance = {
        _name => $name,
        _position => $position,
    };
    bless $instance, $class;
}
sub get_position {
    my $self = shift;
    $self->{_position};
}
sub set_position {
    my $self = shift;
    $self->{_position} = shift;
}
my %_promotion_table = (
    shop_floor => "staff",
    staff => "management",
    astt_manager => "manager",
    manager => "executive",
);
sub promote {
    my $self = shift;
    $self->{_position} =
              $_promotion_table{ $self->{_position} };
```

```
}
sub print {
    my $self = shift;
    print "$self->{_name}  $self->{_position}  ";
}

#------------------  class Manager  ----------------------
package Manager;

    @Manager::ISA = ("Employee");

    sub new {
        my ( $class, $name, $position, $department ) = @_;
        my $instance = Employee->new( $name, $position );
        $instance->{_department} = $department;
        bless $instance, $class;
    }
    sub get_department {
        my $self = shift;
        $self->{_department};
    }
    sub promote {
        my $self = shift;
        die "A manager cannot be promoted beyond 'Executive'"
            if $self->{_position} eq 'executive';
        #call base class's promote by specifying
        #base class name explicitly:
        $self->Employee::promote();
    }
    sub print {
        my $self = shift;
        $self->SUPER::print();
        print "$self->{_department}\n";
    }
```

```
#-------------------------------------------------------------
#                          Test Code
#-------------------------------------------------------------

package main;

my ($position, $dept);

my $emp = Employee->new( "Orpheus", "staff" );

$emp->promote();

$position = $emp->get_position();

print "$position\n";               # management

my $man = Manager->new( "Trillian", "manager", "sales" );

$man->print();     # Trillian  manager   Dept: sales

$man->promote();

$man->print();     # Name: Trillian   Position: executive   Dept: sale

$man->promote();   # A manager cannot be promoted beyond 'Executive'
```

The UNIVERSAL Class

- Every class in Perl inherits implicitly from a class called **UNIVERSAL**. We can therefore say that `UNIVERSAL` is implicitly at the root of every class hierarchy in Perl. `UNIVERSAL` plays the same role that the root class `Object` plays in Java.

- Every class implicitly inherits the following methods from `UNIVERSAL`:
  - isa(class_name)
  - can(method_name)
  - VERSION(need_version)

- Of these, the first two can be invoked by a programmer. The third is invoked by the system automatically if the programmer requests a particular version number for the Perl libraries.

- The `isa()` method above that every class inherits from `UNIVERSAL` can be used to test whether a given object is an instance of a particular class.

```
#!/usr/bin/perl -w

use strict;
```

```
### IsaMethod.pl

#--------------------  class Employee  ---------------------
package Employee;

sub new {
    my ( $class, $name, $position ) = @_;
    my $instance = {
        _name => $name,
        _position => $position,
    };
    bless $instance, $class;
}

#----------------------  class Manager  ---------------------
package Manager;

$Manager::VERSION = 5.2;

@Manager::ISA = ("Employee");

sub new {
    my ( $class, $name, $position, $department ) = @_;
    my $instance = Employee->new( $name, $position );
    $instance->{_department} = $department;
    bless $instance, $class;
}

#-------------------------------------------------------------
#                              test code
#-------------------------------------------------------------

package main;
```

```
my $man = Manager->new( "Trillian", "manager", "sales" );

print $man->isa( 'UNIVERSAL' )  ? "yes\n" : "no\n";     # yes
print $man->isa( 'Manager' )    ? "yes\n" : "no\n";     # yes
print $man->isa( 'Employee' )   ? "yes\n" : "no\n";     # yes
print $man->isa( 'Executive' )  ? "yes\n" : "no\n";     # no
print Manager->isa('UNIVERSAL') ? "yes\n" : "no\n";     # yes
print Manager->isa('Employee')  ? "yes\n" : "no\n";     # yes

@Manager::ISA = ();

print $man->isa( 'UNIVERSAL' )  ? "yes\n" : "no\n";     # yes
print $man->isa( 'Manager' )    ? "yes\n" : "no\n";     # yes
print $man->isa( 'Employee' )   ? "yes\n" : "no\n";     # no
print $man->isa( 'Executive' )  ? "yes\n" : "no\n";     # no
print Manager->isa('UNIVERSAL') ? "yes\n" : "no\n";     # yes
print Manager->isa('Employee')  ? "yes\n" : "no\n";     # no
```

- The other programmer-usable method that is inherited by every class from `UNIVERSAL` is `can()`. This method can be used to test whether a given class supports a particular method, either directly or through inheritance. A call to `can()` returns a reference to the supported method definition.

```
#!/usr/bin/perl -w
use strict;

### CanMethod.pl

package X;
    sub new { bless {}, $_[0] }
```

```
    sub foo { print "X's foo invoked\n"; }

package Y;
    sub new { bless {}, $_[0] }
    sub bar { print "Y's bar invoked\n"; }

package Z;
    @Z::ISA = qw( X Y );
    sub new { bless {}, $_[0] }
    sub baz { print "Z's baz invoked\n"; }

package main;

my $obj = Z->new();

print $obj->can( "foo" ) ? "yes\n" : "no\n";      # yes

print Z->can( "foo" )    ? "yes\n" : "no\n";      # yes

my $which_func = $obj->can( "boo" ) ||
                 $obj->can( "bar" ) ||
                 $obj->can( "baz" );
&$which_func;                           # Y's bar invoked
```

- Finally, the third method that is inherited by every class from `UNIVERSAL` is

```
VERSION( need_version )
```

  This method is invoked directly by the system when a programmer requests that a specified version of a certain class file be

loaded in.

- You can associate a version number with a class file by using the special variable `$VERSION` as shown below for the `Manager` class in the `IsaMethod.pl` script:

```
$Manager::VERSION = 5.2;
```

- Subsequently, when a class file is loaded in through the `use` directive, the load request can be customized to a specific version of the class file by

```
use Manager 5.2;
```

- This causes the invocation of the `VERSION()` method inherited from `UNIVERSAL` through the following command:

```
Manager->VERSION( 5.2 );
```

  which makes sure that the version number is met by the class file.

Summary of How a Method is Searched For in a Class Hierarchy

- Earlier we talked about how a method is searched for in a class hierarchy when the method is invoked on a derived-class object. There our discussion focussed on the direct and the indirect superclasses of a derived-class that are searched for a given method.

- Now we will generalize that discussion to include the class `UNIVERSAL` in the search process.

- We will also address the issue of when exactly a method call is shipped off to `AUTOLOAD()` of the derived-class and the `AUTOLOAD()` of the various direct and the indirect superclasses of a derived class.

- Let's say we have

```
package Z;
@Z::ISA = qw( X Y );
```

- Now a call like

```
Z->foo();
```

will cause a search to be conducted for the method `foo()` in the following order:

1. Perl will first look for `foo()` in the class `Z`. If not found there, continue to next step.

2. Perl will look for `foo()` in the parent class `X`. If not found there, proceed to the next step.

3. Perl will look for `foo()` — and do so recursively in a depth-first manner — in the direct and indirect superclasses of `X`. If not found in any of those superclasses, proceed to the next step.

4. Perl will look for `foo()` in the next base class, `Y`, declared in the `ISA` array above. If not found there, proceed to the next step.

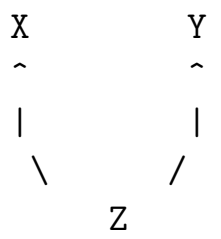
5. Perl will look for `foo()` — and do so recursively in a depth-first manner — in the parent classes of `Y`. If not found there, proceed to the next step.

6. After searching through the programmer-specified superclass hierarchy in the manner described above, Perl will search for the method in the root class `UNIVERSAL`. If not found there, proceed to the next step.

7. Search for an implementation of the `AUTOLOAD()` method in exactly the same manner as outlined in the previous six steps. Ship off the call `foo()` to the first `AUTOLOAD()` implementa-

tion found. If no `AUTOLOAD()` implementation is found at all, throw a run-time exception.

- In the code shown below, we will implement the following hierarchy:



As an aside, note that the objects constructed for each class consist of empty arrays.

```
#!/usr/bin/perl -w
use strict;

### MethodSearch.pl

#----------------------  class  X  ---------------------
package X;
    sub new { bless [], shift }
    sub foo { print "foo of class X called\n" }

#----------------------  class Y  ----------------------
package Y;
    sub new { bless [], shift }
    sub bar { print "bar of class Y called\n"}
    sub AUTOLOAD {
        print "AUTOLOAD of class Y invoked by " .
              "function call $Y::AUTOLOAD\n";
```

```
    }

#--------------------  class Z  -----------------------
package Z;
    @Z::ISA = ('X', 'Y');
    sub new { bless [], shift }

#---------------------  Test Code  ---------------------
package main;

my $zobj = Z->new();

$zobj->foo();        # foo of class X called

$zobj->bar();        # bar of class Y called

$zobj->jazz();
    # AUTOLOAD of class Y invoked by function call Z::jazz
    # AUTOLOAD of class Y invoked by function call Z::DESTROY
```

Abstract Classes and Methods

- Abstract classes play a very important role in OO.

- An abstract class can represent the root interface of a class hierarchy. The concrete classes in the hierarchy can then be the different implementations of the interface, each implementation designed for a different but related purpose.

- As an example, we can have a `Shape` hierarchy. A `Shape` represents an abstract notion. The root class `Shape` may only declare abstract methods such as `area()` and `circumference()`. It would not be able to provide implementations for these methods since, as a pure abstraction, a `Shape` has no geometrical detail. Concrete classes such as `Circle` and `Rectangle` derived from `Shape` would be expected to provide implementations for the abstract methods declared for `Shape`.

- A Perl class can be made abstract by having its constructor throw an exception if it should get invoked by a client of the class. This would prevent a client from constructing an instance of what is supposed to be an abstract class. The same can be done for a method if it is supposed to be abstract.

- When using the above approach, you have to watch out for the fact that should a client inadvertently try to to create an instance of an abstract class, your code will throw a run-time error.

- A measure of protection against such run-time issues related to the use of abstract classes can be addressed by providing the package file (containing the abstract class) with its own `import()` method that ensures that a child concrete class (that will be pulled into a working script after the parent abstract class has been loaded in) has fulfilled its contract with regard to providing the promised implementation code for the methods declared in the abstract class. In general, the `import()` method of a package controls what names declared in the package will be made available without the package qualification syntax in the script into which the package is being imported.