Heap Sort

```
sub pushdown {
    my ($heap, $i) = @_;
    my $size = $heap->[0];
    while($i <= $size/2) {
        my $child = $i * 2;
        if ($child < $size and $heap->[$child]) {
            $child++;
        }
        if ($heap->[$i] >= $heap->[$child]) { last }
        ($heap->[$i], $heap->[$child])
            = ($heap->[$child], $heap->[$i]);
        $i = $child;
    }
}
```

Homework Ten

```
sub pushdown {
   my ($heap, $i) = @_;
   my $size = $heap->[0];
   return if $i >= $size/2;
   my $child = 2 * $i;
   if ($child <= $size and $heap->[$child]){
       $child++;
   }
   if ($heap->[$i] >= $heap->[$child]) { return; }
   ($heap->[$i], $heap->[$child])
       = ($heap->[$child], $heap->[$i]);
   pushdown($heap, $child);
}
```

Programming in Perl

Week Eleven

Object-Oriented Programming and abstract data structures

Jump to first page

Introduction to OOP

- This presentation is a gentle introduction to Object Oriented Perl
  - Laziness on a grand scale
  - Flexibility of Perl's approach to object orientation
  - Moving beyond "one-night-stands" of programming to scaleable programs
  - Having fun!

Topics of Discussion

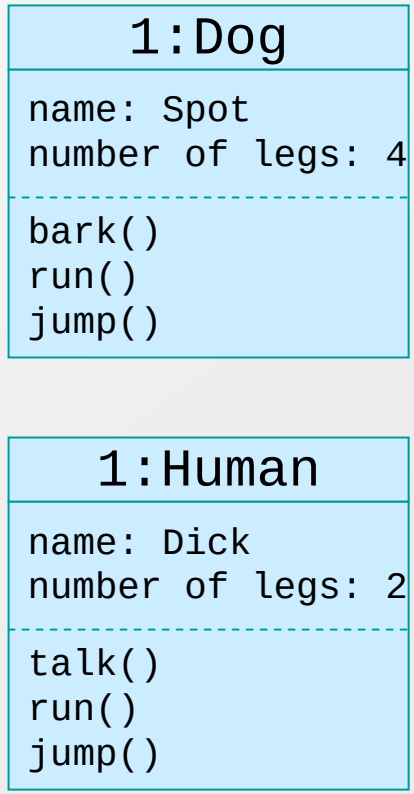
- Differences between Algorithmic and Object-Orientated programming
- What is Object-Orientation?
- What you need to know about Perl
- Getting Started with Object-Oriented Perl
- Other Ways to Handle Perl OO
- Inheritance In Perl
- Polymorphism In Perl
- Example

Differences in Algorithmic and Object-Oriented Programming

- In algorithmic programming, we focus on how to get something done
  - The action we want done is more important than the data
  - We use verbs to describe what we want done
  - We want to sort, transform, edit, run, bark, etc.
- In object-oriented programming, we focus on what we are working on
  - We work on dividing and structuring the data more that the actions
  - We use nouns to describe what we are working on
  - We work with heaps, windows, ATMs, Dogs, Cats, etc.

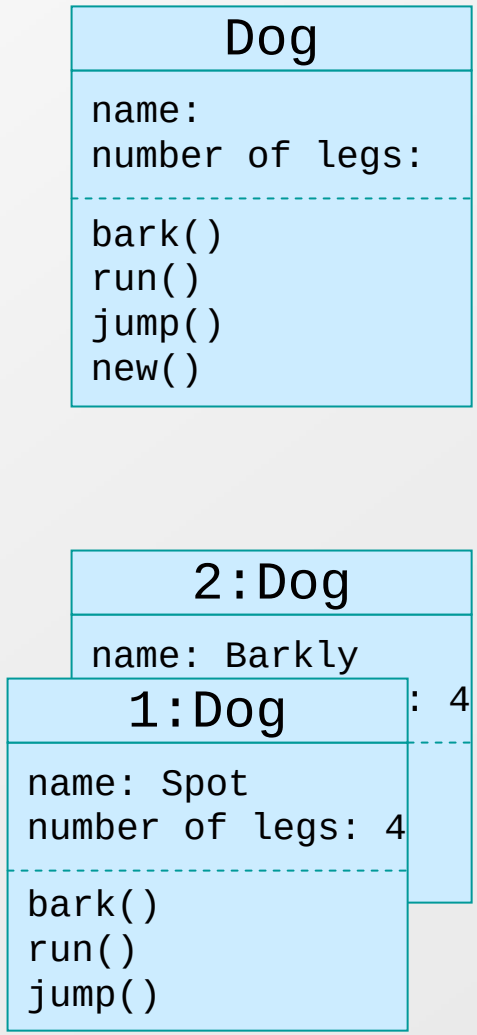
What is Object-Orientation?

- An Object is a container for data
  - The data is the object's attribute values
- Objects are an abstraction that hides the complexity of the data
  - Encapsulation make the object attributes not directly assessable to the rest of the program
  - Access to the attributes is done through methods that are assessable to the rest of the program
  - *Inheritance* and *polymorphism* allow you abstract even further



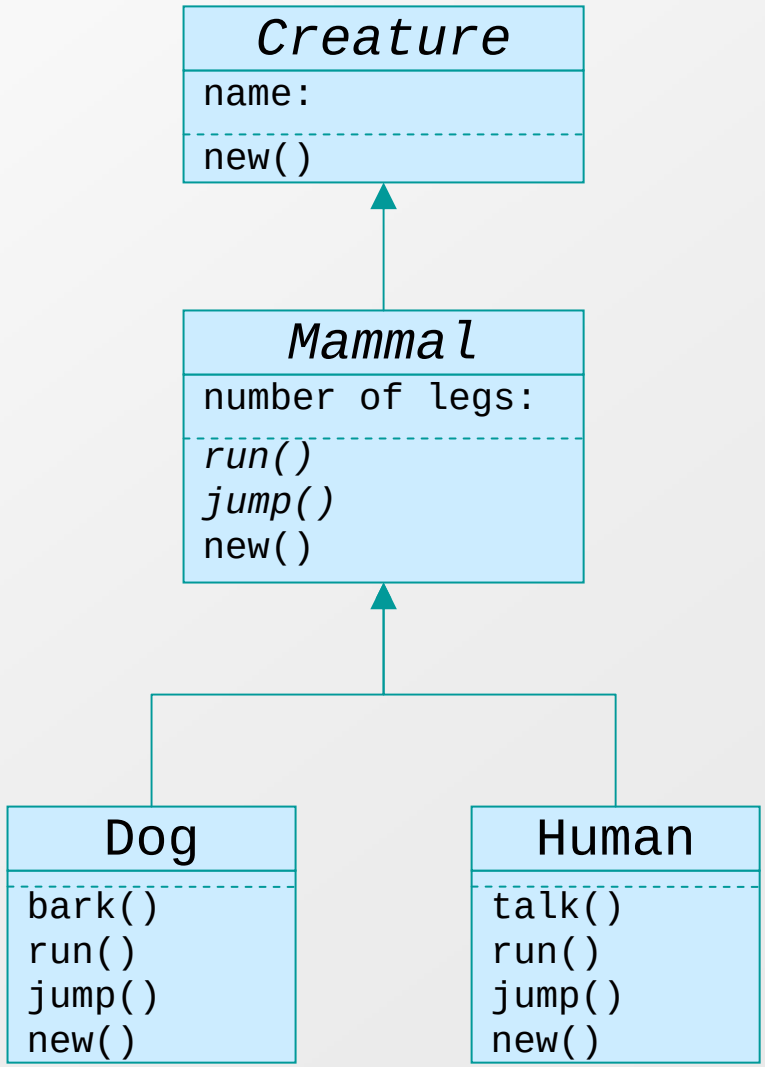
Class

- *Classes* are a blueprint for a set of objects. They define the properties of the object:
  - A common interface, i.e.: the methods, for all objects belonging to the class.
  - The implementation of the methods.
  - A description of the data the object holds.
- Objects are an *instance* of a class
  - You create an object from a class by calling a creator method



Inheritance

- We can create a tree of classes, where the properties from a parent are inherited by the child
- As we go up the inheritance tree, the properties become more general.
  - A Human is-a Mammal is-a Creature is-a Living Thing
- If a class is only used as a parent class (no object will be created from it,) the class is called an *abstract class*.



Polymorphism

- It’s possible to invoke identical methods (this is called sending a *message*) of two, or more, objects and have different actions result. This behavior is called *polymorphism*.
- If a parent class,in a hierarchy, defines a method, that method is available to all of the children classes. This is *hierarchy polymorphism*.
- *Interface polymorphism* is where unrelated classes provide the same method.

What you need to know about Perl

- Perl's data types and how to manipulate them:
  - Scalar– assigning to and accessing, the undefined (`undef`) value
  - Array– LISTs, accessing, slicing, iterating over, using an array as a stack
  - Hash– keys and values, accessing, iterating over
- Control structures:
  - `if`/`then`/`else` (`unless`)
  - `while` (`until`) loops, `for`/`foreach` loops `next`, `last`, `redo`

What you need to know about Perl

- User defined subroutines
  - Defining with the functions with the `sub` operator
  - Calling subroutines `doit($acton);`
  - The last value evaluated is the return value (you can also use the `return` operator)
  - Passing arguments to a subroutine using the `@_` array (and that the values in `@_` are aliases, not copies, to the values passed to the subroutine)
  - Localizing variables with `my`

What you need to know about Perl

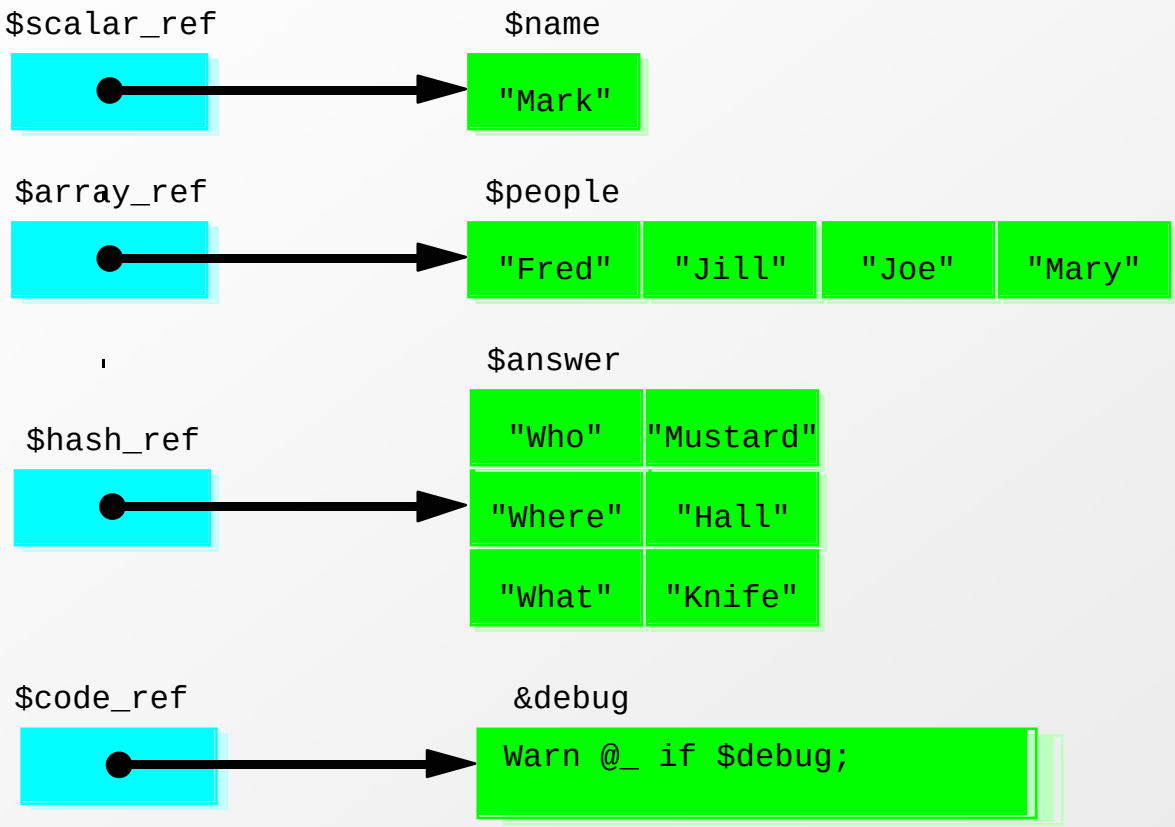
- Calling Contexts of subroutines
  - Void context– `listdir(@files);`
  - Scalar context– `$listed = listdir(@files);`
  - List context– `@missing = listdir(@files);`
- Determining the calling context with `wantarray()`

```
sub listdir {
    # Guts of listdir
    return @missing files if wantarray;
    return $list_counted if defined wantarray;
}
```

What you need to know about Perl

- References
  - References are a way to use a variable indirectly
  - A reference “points” to the value of variable without knowing the name of the variable.
  - A reference is always contained in a scalar variable (or a scalar element in an array or hash)
  - You can point to any type of data that Perl knows about

References


$scalar_ref
$name
"Mark"
$array_ref
$people
"Fred"
"Jill"
"Joe"
"Mary"
$answer
$hash_ref
"Who"
"Mustard"
"Where"
"Hall"
"What"
"Knife"
$code_ref
&debug
Warn @_ if $debug;

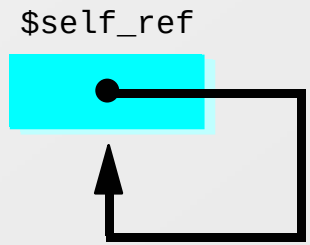
References

- Named references point to a variable that has been already been declared in the script
- You can create references to variables by adding a backslash (“\”) before the variable

```
$scalar_ref = \$name;
$array_ref = \@people;
$hash_ref = \%answer;
$code_ref = \&debug;
```

- You can even create a reference to itself!

```
$self_ref = \$self_ref;
```



References

- The Arrow (“->”) Operator
  - There are two ways to access a value pointed to by a reference.
    - The confusing way: de-referencing

```
${$array_ref}[2] = ${$hash_ref}{’Who’};
```

    - The simpler way: the arrow operator

```
$array_ref->[2] = $hash_ref->{’Who’};
```

  - The arrow operator only works with arrays or hashes (and methods).
  - To get to scalars (and other things) you must use the de-referencing syntax

```
$$scalar_ref = "Fred";
```

References

- Identifying a Referent
  - You can identify what kind of reference you have by using the `ref()`

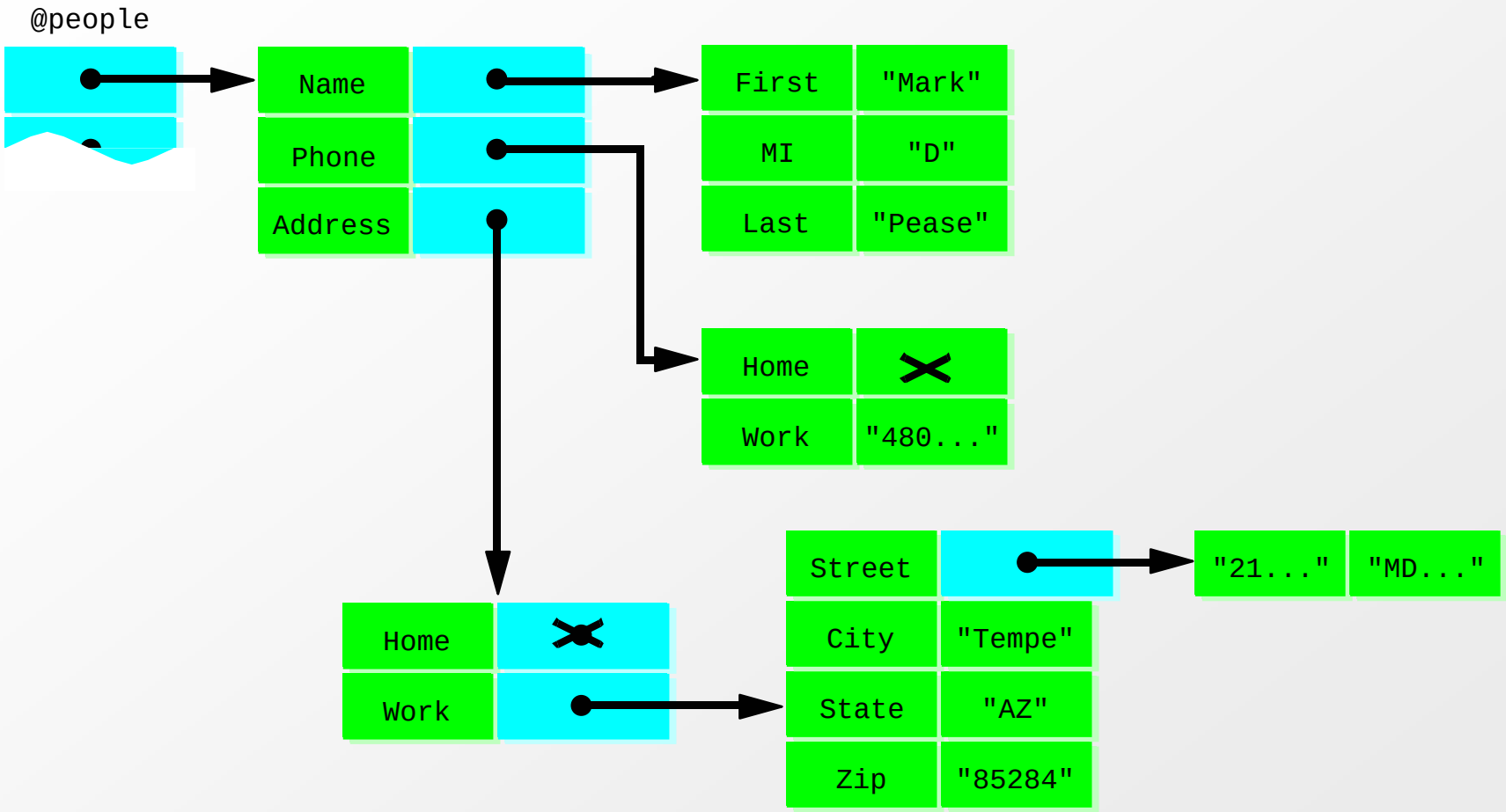
Value of Reference	`ref()` **returns**
`scalar reference`	`"SCALAR"`
`array reference`	`"ARRAY"`
`hash reference`	`"HASH"`
`subroutine reference`	`"CODE"`
`filehandle reference`	`"IO" or "IO::Handle"`
`typeglob reference`	`"GLOB"`
`Regular expression reference`	`"Regexp"`
`reference reference`	`"REF"`
`a scalar value`	`undef`

References

- You can build complex data structures using references to arrays and hashes.

```
$people[0] = {
   Name      =>      { First => "Mark", MI => "D", Last => "Pease" },
   Address   =>      {
                     Work =>
                     {
                              Street => [
                                       ’2100 E. Elliot Rd’,
                                       ’MD EL701’
                              ],
                              City => ’Tempe’,
                              State => ’AZ’,
                              Zip = 85282
                     },
                     Home => undef
             },
   Phone     =>      { Work => ’480-413-8191’, Home => undef }
}
```

References


@people
Name
Phone
Address
First
"Mark"
MI
"D"
Last
"Pease"
Home
Work
"480..."
Home
Work
Street
"21..."
"MD..."
City
"Tempe"
State
"AZ"
Zip
"85284"

What you need to know about Perl

- Packages
  - Packages allow you to create namespaces to separate like named variables and subroutines
  - You change name spaces by using the package operator

    ```
    package call;
    ```

  - By default, Perl uses the main package
  - You can access the variables or subroutines from within the package normally
  - To access a variable or subroutine in a package from another package, put ‘`::`” between the package name and the symbol.

    ```
    Call::number(“480-413-8191”);
    ```

Getting Started With Object Oriented Perl

- There are Three little Rules to OOP in Perl
  - Rule 1: To create a class, build a package
  - Rule 2: To create a method, write a subroutine
  - Rule 3: To create an object, bless a referent

Rule 1: To create a class, build a package

- Perl packages already have a number of class-like features
  - They collect related code together
  - They distinguish that code from unrelated code
  - They provide a separate namespace within the program, which keeps subroutine names from clashing with those in other packages
  - They have a name, which can be used to identify data and subroutines defined in the package
- Packages provide encapsulation
- Perl knows how to handle modules, so you should put classes in modules (`class.pm`)

Rule 2: To create a method, write a subroutine

- Subroutines are associated with a package in the same way that methods must be associated with class
- When a subroutine is called as a method, the first argument is always a reference to the object that called it

```
sub print_name {
    my $self = shift;
    # Do something with the object.
}
```

Rule 3: To create an object, bless a referent

- Any Perl data type can be converted to an object
  - You don't need a special definition of a recordlike data structure, like some other languages
- To create an object, you bind a reference to some data to a class using the bless operator

```
$person = { … };
bless $person, "Person";
```

- The ref operator would return "HASH" before the bless, and would return "Person" after.
- The object is still a hash, but has been marked as an object

Calling methods

- You use the “`->`” operator to call methods much like you use it to access data through a reference

```
$person->print_name;
```

- The object is passed as the first argument of the method, and any others are added after

Making an object creator

- Bless returns the blessed reference, so you can write a subroutine that will create a new object
- When using the "`->`" operator, if you use a package name instead of a reference, the package name is passed as the first argument

```
sub new {
    my $class = shift;
    my $obj = {
        # Process the arguments to create the data
    };
    return bless $obj, $class;
}

$person = Person->new(…);
```

Accessing data from an object

- Because the object is a reference to some Perl data, you can use reference notation to access that data

```
sub print_name {
    $self = shift;
    print “$self->{Name}->{First} $self->{Name}->{Last}\n”;
}
```

Inheritance

- Perl also provides a way to do inheritance
- Because inheritance is a “is-a” relationship, Perl uses a special package array variable `@ISA` to show inheritance.

```
package Employee;
@ISA = qw( Person );
```

- Inheritance is simple in Perl: If you can’t find the method requested in an object’s class (the package), look for it in the classes that the object’s class inherits from

Polymorphism

- In Perl, every method of every class is potentially polymorphic because of the way that methods are automatically dispatched up the class hierarchy
  - If a method has been defined in a package up the hierarchy, then it is available to all of its children
- You can use interface polymorphism by just creating a method, in all the classes that must be polymorphic, that has the same name and takes the same arguments

That the beginning!

- There is lots more to Object Oriented Perl
  - Using data types, other than hashes, for objects.
  - Multiple Inheritance
  - Inheriting constructors
  - Operator overloading
  - “Real” Encapsulation
  - etc.
- Get a copy of Damian Conway’s “Object Oriented Perl” from Manning, for lot more info.

Example: A Stack

- A stack is a standard data structure use in lots of ways
- The “Stack” is like a stack of books
  - You can put a new book on the top of the stack
  - You can take a book off of the stack
- For the data structure, you can:
  - `push(item)`
  - `pop()`
  - `is_empty()`
  - `is_full()`

Stack.pm

```
package Stack;
use Carp;

sub new {
    my $type = shift;
    my $class = ref($type) || $type;
    my $max_size = shift;
    my $self = [$max_size];
    return bless $self, $class;
}
```

Stack.pm

```
# $stack->is_empty();  returns true if the stack is empty
sub is_empty {
    my $self = shift;
    return !$#$self;
}

# $stack->is_full(); return true if the stack is full
sub is_full {
    my $self = shift;
    return 0 unless $self->[0];
    return ($#$self == $self->[0]);
}
```

Stack.pm

```
# $stack->push($item); pushes $item onto stack if stack not full
sub push {
    my $self = shift;
    my $item = shift;
    if ($self->is_full()) {
        carp “Stack is full:”;
        return;
    }
    push @$self, $item;
}
```

Stack.pm

```
# $stack->pop(); pops the top item from the stack if not empty
sub pop {
    my $self = shift;
    if ($self->is_empty()) {
        carp “Stack is empty:”;
        return;
    }
    return pop @$self;
}
```

Stack.pm

```
# $stack->top(); returns the value of the top element if not empty
sub top {
    my $self = shift;
    if ($self->is_empty()) {
        carp “Stack is empty:”;
        return;
    }
    return $self->[$#$self];
}
1;
__END__
```

Using Stack.pm

```
#!/usr/bin/perl -w
use strict;
use Stack;
my $st = Stack->new(4);
# test push() to overflow
for(3,5,2,9,11) {
    print “pushing: $_\n” if $st->push($_);
}
print “popped: “, $st->pop(), “\n”;
print “pushed: 42” if $st->push(42);
print “top is: “, $st->pop();

#test pop to underflow
for(1..5) {
    print ‘popped: ‘, $st->pop(), “\n”;
}
print $st->top(), “\n”;
```

```
pushing: 3
pushing: 5
pushing: 2
pushing: 9
Stack is full: at stack.pl
line 7
popped: 9
pushed: 42
top is: 42
popped: 42
popped: 2
popped: 5
popped: 3
Stack is empty: at
stack.pl line 16
popped:
Stack is empty: at
stack.pl line 18
```