Core Design Concepts Discussed:

Syntax

Analysis

Evaluation

Object-Oriented Programming

A Brief Introduction

Harley Eades III

In this lecture we are going to take a brief look at how object oriented programming is supported by OCaml. But, first…

What is OCaml?

Core Design Concepts:

Syntax

An object oriented, imperative, functional programming language.

What is OCaml? In fact, it supports objects, imperative programming, and as we have seen functional programming. Furthermore, OCaml…

What is OCaml?

Core Design Concepts:

Syntax

An object oriented, imperative, functional programming language.

OCaml mixes all of these paradigms together.

Mixes all of these paradigms together. We can use them together if we wish to. Now…

What is OCaml?

Core Design Concepts:

Syntax

An object oriented, imperative, <u>functional programming language</u>.

OCaml mixes all of these paradigms together.

We’ve been focusing on functional programming, because that is a new concept for most, if not all, of you…

What is OCaml?

Core Design Concepts:

Syntax

An object oriented, <u>imperative</u>, functional programming language.

OCaml mixes all of these paradigms together.

OCaml supports imperative programming as well where we can use references, pointers, and state. So far we've been using its immutable support, because that is also a new style of programming for most of you.

What is OCaml?

Core Design Concepts:

Syntax

An <u>object oriented</u>, imperative, functional programming language.

OCaml mixes all of these paradigms together.

However, in this lecture we are going to concentrate on objects, but this will require…

What is OCaml?

Core Design Concepts:

Syntax

An object oriented, imperative, functional programming language.

OCaml mixes all of these paradigms together.

Imperative programming as well, since objects have a notion of state by definition.

Class Definitions

Core Design Concepts:

Syntax

```
class name = object (self) … end
```

This is the syntax for defining a class. We give it a name, and then declare that we are building an object where we pass `self` into the object for self references. Then…

Class Definitions

Core Design Concepts: Syntax

```
class stack_of_ints =
  object (self)
    val mutable the_list = ([] : int list)
     …
  end;;
```

We use `val mutable` to declare fields (instance variables). At that point, …

Class Definitions

Core Design Concepts:

Syntax

```
class stack_of_ints =
  object (self)
    val mutable the_list = ([] : int list)
    method push x = …
    method pop = …
    method peek = …
    method size = …
  end;;
```

each method is just a typical OCaml function that is allowed to use the field variables, but must be defined using the `method` keyword. Let's take a closer look at each of the methods…

Class Definitions

Core Design Concepts:

Syntax

```
class stack_of_ints =
  object (self)
    val mutable the_list = ([] : int list)
    method push x = the_list <- Cons(x, the_list)
    method pop = …
    method peek = …
    method size = …
  end;;
```

The push method uses the fact that a list is essentially a stack where cons pushes a new element on the front (or top) of the list.

Class Definitions

Core Design Concepts:

Syntax

```
class stack_of_ints =
  object (self)
    val mutable the_list = ([] : int list)
    method pop =
      let result = head the_list in
      the_list <- tail the_list;
      result
    method push x = …
    method peek = …
    method size = …
  end;;
```

Pop uses head to extract the head of the list, the top most element of the stack, then we reassign the_list to be it's tail removing the head of the list. Then return the result. This is a nice example of imperative programming in OCaml.

Class Definitions

Core Design Concepts:

Syntax

```
class stack_of_ints =
  object (self)
    val mutable the_list = ([] : int list)
    method push x = …
    method pop = …
    method peek = head the_list
    method size = …
  end;;
```

Peek is just the head of the list without reassigning the_list to its tail. Notice we do not reassign to the the_list resulting in the_list remaining the same.

Class Definitions

Core Design Concepts:

Syntax

```
class stack_of_ints =
  object (self)
    val mutable the_list = ([] : int list)
    method push x = …
    method pop = …
    method peek = …
    method size = length the_list
  end;;
```

Finally, size simply calculates the length of the_list.

Class Definitions

Core Design Concepts:

Syntax

```
class stack_of_ints =
  object (self) …
end;;
class stack_of_ints :
  object
    val mutable the_list : int list
    method peek : int
    method pop : int
    method push : int -> unit
    method size : int
  end
```

Every class must also declare its signature. That lists all of the types of the fields and methods.

Accessing fields and methods

Core Design Concepts:

Syntax

```
# let s = new stack_of_ints;;
val s : stack_of_ints = <obj>
```

We create new objects of our class using the `new` keyword. Then…

Accessing fields and methods

Core Design Concepts:

Syntax

```
s#fieldName
s#methodName
```

We can access fields and methods using the `#` operator.

Accessing fields and methods

Core Design Concepts:

Syntax

```
# for i = 1 to 10 do
    s#push i
  done;;
- : unit = ()
# while s#size > 0 do
    Printf.printf "Popped %d off the stack.\n" s#pop
  done;;
…
Popped 10 off the stack.
Popped 9 off the stack.
Popped 8 off the stack.
- : unit = ()
```

Here is an example of using loops in OCaml, and the class we implemented.