

# CS 421 Lecture 19: Functional Programming in Object-Oriented Languages

---

- Announcements
- Lecture outline
  - Multi-paradigm programming languages
  - Functional programming in OOP
  - Function objects
  - Examples
    - Java
    - C++
- Based on slides developed by Dongyun Jin

# Announcements

---

- MP7 has been posted
- Midterm grades will be up soon
  - Questions about your grade

# Objectives

---

- Multi-Paradigm Programming Language
  - Other paradigms in a Programming Language
- Functional Programming in OOP
  - Function Object
- Examples
  - Function Objects in Java
  - Function Objects in C++
  - Parser Combinators in Java

# Multi-paradigm PL

---

- Programming Languages that support more than one programming paradigm
- Java
  - Imperative, Generic, Reflective, Object-Oriented
- C++
  - Imperative, Generic, Object-Oriented
- Ocaml
  - Functional, Imperative, Generic, Object-Oriented
- Why?
  - No one paradigm solves all problems in the easiest or most efficient way.

# Other paradigms in a PL

---

- In a programming language
  - Some paradigms are supported fundamentally
  - Other paradigms can be simulated by using language features
- If a programming language is Turing-complete, any paradigm can be simulated in the language
- Problem
  - Conciseness
  - Efficiency
  - Difficulty

# Functional programming in OOP

---

- Functional Programming
  - No Side Effects
  - Dynamic Memory Allocation
  - Recursion
  - Higher-Order Functions
  - Lazy Evaluation
- In Object-Oriented Language,
  - Naturally Accomplishable
  - Useful

# Function object

---

- Function Object
  - Also called Functor or Functionoid
  - A programming construct allowing an object to be invoked or called as if it were an ordinary function

## In Ocaml

```
map (fun x -> x+1) [1;2;3;4]
```

inside of map:

```
(fun x -> x+1) (1)
```

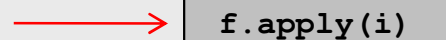
```
(fun x -> x+1) (2)
```

```
...
```

## In OO language

```
List map(IntFun f, List l);
```

inside of map:

```
j = f(i); 
```

But f is an object.

```
Class IntFun{...}
```

# Purpose of function objects

---

- More Expressiveness
  - Sometimes simpler and more convenient than other approaches
  - High-Order Functions
- Resilient to Design Changes
  - We don't need to change the interface
  - Create New Functionality without writing new functions



# Function objects in Java

---

- Function object interface and definition

```
interface IntFun {  
    int apply(int x);  
}  
  
class Incr implements IntFun {  
    ... // blah blah  
}
```

## In Ocaml

```
let f x = x + 1;;  
  
map f A;;
```

- Inner class

```
map(  
    new IntFun {  
        int apply(int x){return x+1;}}  
    , A);
```

## In Ocaml

```
map (fun x -> x+1) A;;
```

# Function objects in Java

---

- Function interface example 1

```
interface IntFun {
    int apply(int x);
}

... //some classes and blah

int[] map(IntFun f, int A[]) {
    int B[] = new int[A.length];
    for(int i = 0, i < A.length; i++)
        B[i] = f.apply(A[i]);
    return B;
}
```

```
class Incr implements IntFun {
    int apply(int x) {
        return x + 1;
    }
}

... //some classes and blah

int A[] = new int[30];
...
int C[] = map(new Incr(), A);
```

# Function objects in Java

---

- Function interface example 2

```
interface IntBinaryPred {
    boolean apply(int x, int y);
}

class GT implements IntBinaryPred {
    boolean apply(int x, int y) {
        return x > y;
    }
}
```

```
Interface BoolBinaryPred {
    boolean apply(boolean x, boolean y);
}

class And implements BoolBinaryPred {
    boolean apply(boolean x, boolean y)
    {
        return x && y;
    }
}
```

# Function objects in Java

---

- Inner class

```
interface IntBinaryPred {
    boolean apply(int x, int y);
}

class GT implements IntBinaryPred {
    boolean apply(int x, int y) {
        return x > y;
    }
}

...
CompareArray(new GT(), A, B);
```

```
Interface IntBinaryPred {
    boolean apply(int x, int y);
}

...
CompareArray(
    new IntBinaryPred {
        boolean apply(int x, int y){
            return x > y;}}
    , A, B);
```

# Function objects in Java

---

- Real example – Comparator

```
List<String> list = Arrays.asList(new String[] {
    "10", "1", "20", "11", "21", "12"
});

Collections.sort(list,
    new Comparator<String>() {
        public int compare(String o1, String o2) {
            return Integer.valueOf(o1).compareTo(Integer.valueOf(o2));
        }
    }
);
```

# Function objects in Java

---

- Higher-order function objects

```
interface IntFun {
    int apply(int x);
}

IntFun combine-add(IntFun f, IntFun g) {
    return new IntFun {
        int apply(int x) {
            return f.apply(x) + g.apply(x);
        }
    };
}
```

In Ocaml:

```
let combine-add f g = fun x -> (f x) + (g x);;
```

# Function objects in Java

---

- Higher-order function objects

```
interface IntFun {
    int apply(int x);
}

interface IntFun2 {
    int apply(int x, int y);
}

IntFun compose2 (IntFun2 f, IntFun g, IntFun h) {
    return new IntFun {
        int apply(int x) {
            return f.apply(g.apply(x), h.apply(x));
        }
    };
}
```

In Ocaml:

```
let compose2 f g h = fun x -> f(g x, h x)
```

# Function objects in C++

---

- Operator overloading

```
Incr incr = new Incr();

a = incr(3);

class Incr{
public:
    int operator() (int x) {
        return x + 1;
    }
}
```



# Function objects in C++

---

- Operator overloading example 1

```
class IntFun {
public:
    int operator() (int x){
        // return blah ...
    }
}

template<class IntFun>
int* map(IntFun f, int *A, int n) {
    int *C = new int[n];
    for(int i = 0; i < n; i++)
        C[i] = f ( A[i] );
    return C;
}
```

# Function objects in C++

---

- Operator overloading example 2

```
class GT {  
    public:  
    bool operator() (int x, int y) {  
        return x > y;  
    }  
}  
  
class AND {  
    public:  
    bool operator() (bool x, bool y) {  
        return x && y;  
    }  
}
```

# Function objects in C++

---

- Higher-order functors – example 1

```
template<class IntFun>
class Combine_Add {
public:
    IntFun f, g;
    Combine_Add(IntFun f, IntFun g) {
        this.f = f;
        this.g = g;
    }
    int operator() (int x) {
        return f(x) + g(x);
    }
}
```

**In Ocaml:**

```
let combine-add f g = fun x -> (f x) + (g x);;
```

# Function objects in C++

---

- Higher-order functors – example 2

```
template<class IntFun>
class Funmod {
public:
    IntFun f; int x, y;
    Funmod(IntFun f, int x, int y) {
        this.f = f; this.x = x; this.y = y;
    }
    int operator() (int z) {
        if(x == z)
            return y;
        else
            return f(z);
    }
}
```

## In Ocaml:

```
let funmod f x y =
    fun z -> if x = z then y else f z;
```

# Parser combinators in Java

```
interface Parser {
    ArrayList apply(ArrayList cl);
}

Parser token(String s) {
    return new Parser {
        ArrayList apply(ArrayList cl) {
            ArrayList cl2;
            if (cl == null || cl.size() == 0)
                return null;
            if (s.compareTo((String)cl.get(0)) == 0) {
                cl2 = cl.clone();
                cl2.remove(0);
                return cl2;
            }
            return null;
        }
    };
}

Parse parserx = token("x");
```

## In Ocaml:

```
let token s = fun cl ->
  if cl = [] then None
  else if s = hd cl
    then Some(tl cl)
    else None;;

let parserx = token 'x';;
```

# Parser combinators in Java

```
Parser plusplus(Parser p, Parser q) {
    return new Parser {
        ArrayList apply(ArrayList cl) {
            ArrayList cl2 = p.apply(cl);
            if(cl2 == null) return null;
            else return q.apply(cl2);
        }
    };
}

Parse parserxy = plusplus(token("x"), token("y"));

Parser or(Parser p, Parser q) {
    return new Parser {
        ArrayList apply(ArrayList cl){
            ArrayList cl2 = p.apply(cl);
            if(cl2 == null) return q.apply(cl);
            else return cl2;
        }
    };
}

Parse parserxyorz = or(parserxy, token("z"));
```

## In Ocaml:

```
let (++) p q = fun cl ->
    match p cl with None -> None
    | Some cl' -> q cl';;

let parsexy = token 'x'
              ++ token 'y';;

let (||) p q = fun cl ->
    match p cl with None -> q cl
    | Some cl' -> Some cl'

let parsexyorz = parsexy
                  || token 'z';;
```

# Parser combinators in Java

---

- $A \rightarrow aA \mid b$

```
Parser parserA = or(plusplus(token("a"), parserA), token("b"));
```

Is it correct?

```
Parser parserwrap =
```

```
    new Parser {
```

```
        Parser recParser = null;
```

```
        ArrayList apply(ArrayList cl) {
```

```
            if(recParser != null) return recParser.apply(cl);
```

```
            return null;
```

```
    }};
```

```
Parser parserA = or(plusplus(token("a"), parserwrap), token("b"));
```

```
parserwrap.recParser = parserA;
```