

# Lecture 26 - Higher-order functions in object-oriented languages

- **Function objects**
- **Java examples**
- **C++ examples**
- **Higher-order functions in the Standard Template Library**

# Function objects

- Also called “functors” (e.g. in C++)
- In Java, can be an object that defines one method, called `apply`
- Can use function objects as if they were functions in a functional language
  - Write `f.apply(...)` instead of `f(...)`

# Purposes of function objects

- **More Expressiveness**
  - Sometimes simpler and more convenient than other approaches
  - Higher-order functions
- **Resilience to design changes**
  - We don't need to change the interface
  - Create new functionality without writing new functions

# Example: increment object

```
class Succ {  
    int apply(int x) {  
        return x + 1;  
    }  
}  
  
...  
Succ incr = new Succ();  
... incr.apply(3) ...
```

# Example: Plus class – building function objects

```
class Plus {
    int i;
    public Plus(int i) {
        this.i = i;
    }
    int apply(int x) {
        return x + i;
    }
}
...
Plus incr = new Plus(1);
... incr.apply(3) ...
Plus add4 = new Plus(4);
... add4.apply(3) ...
```

# Higher-order functions: map

```
void map(Plus p, int[] A) {  
    for (int i = 0; i < A.length; i++) {  
        A[i] = p.apply(A[i]);  
    }  
}  
  
...  
map (new Plus(3), w); // add 3 to each element of w
```

# Interfaces

- This version of `map` can only be used to increment elements of an array by some value
- To define `map` so that it can apply other functions, we need an interface to give the type of any integer-to-integer function object

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

- Need to declare `Plus` to say that it implements this interface. Then we can redefine `map`.

# Interfaces (cont.)

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

```
class Plus implements IntFun {  
    ... same as above ...  
}
```

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



# Interfaces (cont.)

- Now, we can define other integer-valued functions as function objects and use them in map:

```
class MultBy implements IntFun {
    int m;
    public MultBy(int m) {
        this.m = m;
    }
    public int apply(int x) {
        return m * x;
    }
}
... map(new MultBy(5), A);
```

# Interfaces (cont.)

- We can also have generic unary functions

```
interface Function<A, B> {  
    B apply(A x);  
}
```

```
class Plus implements Function<Integer, Integer> {  
    ... same as above ...  
}
```

```
<V> void map(Function<V, V> f, V[] A) {  
    for (int i = 0; i < A.length; i++) {  
        A[i] = f.apply(A[i]);  
    }  
}
```

# Higher-order functions: filter

```
interface IntPredicate {
    public boolean apply(int input);
}

... List<Integer> filter(IntPredicate pred, List<Integer> A) {
    result = new ArrayList<Integer>();
    for (Integer a : A) {
        if (pred.apply(a)) {
            result.add(a);
        }
    }
    return result;
}

class IsEven implements IntPredicate {
    public boolean apply(int input) {
        return input % 2 == 0;
    }
}

... filter(new IsEven(), A); // Returns a list of all even integers in
```

# More higher-order functions

- Can define classes that create functions from other functions, just as in OCaml.

```
class ComposeFuns implements IntFun {
    IntFun f, g;
    public ComposeFuns(IntFun f, IntFun g) {
        this.f = f;
        this.g = g;
    }
    public int apply(int x) {
        return f.apply(g.apply(x));
    }
}
...
map(new ComposeFuns(new Plus(3), new MultBy(5)), A);
```

# More examples of higher-order functions

```
class MaxOfFuns implements IntFun {
    IntFun[] funs;
    public MaxOfFuns (IntFun[] funs) { this.funs = funs; }
    public int apply(int x) {
        int max = funs[0].apply(x);
        for (int i = 1; i < funs.length; i++) {
            int m = funs[i].apply(x);
            max = max > m ? max : m;
        }
        return max;
    }
}
```

```
IntFun[] flis = {new Plus(3), new MultBy(5), ...};
map(new MaxOfFuns(flis), A);
```

# Anonymous inner classes

- Given an interface `IntFun`, we can define classes that implement it anonymously. These are defined within methods (of any class) using the syntax:
  - `new IntFun() { int apply(int x) { ... } }`
- For example
  - **Java:** `map(new IntFun { int apply(x) { return x * x; } }, A);`
  - **OCaml:** `map (fun x -> x * x) A`

# Aside: Event-driven programming in Java

- Inner classes are often used in Java to provide “callbacks”
  - methods that are called when an external event (e.g. mouseclick) occurs, e.g.

```
interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
...
Button b = new Button("Blue");
...
b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        src.setBackground(Color.Blue);
    }
});
```

# Function objects in C++

- The idea of function objects can be used in any OO language
  - Although type checking issues may be different
- C++ has an operator overloading feature. We can even overload (), or function application. In C++, a function object is sometimes defined as an object that overloads function application.

```
class Succ {
    int operator() (int x) {
        return x + 1;
    }
};
Succ succ = new Succ();
a = succ(3);
```



# Function objects in C++

- C++ does not have interfaces, but the type of the function object can be given as a template argument:

```
template<class IntFun>
void map(IntFun f; int *A; int n) {
    for (int i = 0; i < n; i++) {
        A[i] = f(A[i]);
    }
}
map(Succ(), arr, length);
```

# Higher-order functions in C++

```
template<class IntFun>
class Compose {
    IntFun f, g;
    Compose(IntFun f, IntFun g) {
        this->f = f;
        this->g = g;
    }
    int operator() (int x) {
        return f(g(x));
    }
};
```

# Higher-order functions in C++ (contd.)

```
template<class IntFun>
class Curry {
    IntFun f;
    int x;
    Curry(IntFun f, int x) {
        this->f = f;
        this->x = x;
    }
    int operator() (int y) {
        return f(x, y);
    }
};
```

- This is exactly analogous to using a curried function in OCaml and partially applying it.

# Higher-order functions in the Standard Template Library (STL)

- A large section of the STL spec is devoted to functors. Their use is complicated by type-checking issues we have ignored in the above, but the basic ideas are as we have discussed.
- The STL provides some functions that construct function objects from other function objects. We give one example.
- `bind1st`: Given a function object that overloads `operator()` as a two-argument function, and given a value for the first argument, this produces a function object that overloads `operator()` as a one-argument function. E.g.

```
transform(v1.begin(), v1.end(), v2.begin(),  
         bind1st(Plus(), 3.0));
```

# Higher-order functions in the Standard Template Library (STL) (cont.)

- Another place function objects are used is as arguments to functions like `sort`

```
sort(collection_start, collection_end, comparator)
```

where `comparator` is an object that redefined `operator()` as a `bool`-valued function of two arguments...

# Higher-order functions in the Standard Template Library (STL) (cont.)

```
class GT {
    bool operator() (double x, double y) {
        return x > y;
    }
};

... vector<double> u;
    sort(u.begin(), u.end(), GT());
```

**This provides flexibility. We could define another comparator:**

```
class GTMag {
    bool operator() (double x, double y) {
        return fabs(x) > fabs(y);
    }
};
```

# Comparators: another example

```
struct Point {  
    int x, y;  
};
```

```
class CompareYX {  
    bool operator() (const Point &p1, const Point &p2) {  
        return p1.y < p2.y || (p1.x == p2.x && p1.x < p2.x);  
    }  
};
```

```
... // Sorts by y coordinate, breaking ties by x-coordinate  
    sort(points.begin(), points.end(), CompareYX());
```

# Template parameters

- For the record, we have simplified the above by omitting required type declarations. GT is actually

```
class GT : public binary_function<double, double, bool> {  
    bool operator() (double x, double y) { return x > y; }  
};
```

- The `binary_function` doesn't have any parameters, but just provides typedefs for the type parameters above; this goes beyond the scope of today's discussion. In any case, `sort` requires a comparator which is a subclass of `binary_function`, which means just that it overloads `operator()` as a two-argument function.



# Wrap up

- Today we discussed how to use higher-order functions in object oriented languages, and how such functional programming techniques add expressiveness. More specifically, we looked at function objects in Java and C++, and how they can be leveraged in the standard template library.
- On Tuesday
  - Wrap-up