Functors and Lambdas… λ

- Functions
- Functors are an OO idiom to wrap a function in an object
  - ≈ OO functions
  - But, syntactic overhead
- So want simpler way to describe (define) functions
  - ⇨ Lambda
  - Standard definition since LISP (pre-LISP!)
  - λ x . x+1 ≡ f(x) = x+1 ≡ x -> x+1
- So how to do both, describe as lambda, but stay within OO paradigm?
- Aha! → convert λ in to OO
  - λ ⇨ SAM
    (Single Argument Method)
  - so basically, we write a λ, compiler converts into a functor
  - Which functor type?
    - Infer by type of context expression
- So what effect on patterns?
  - Strategy
  - Observer
  - Command
  - Iterators
  - …
- SAM ⇨ *@FunctionalInterface*
  - Several convenience types provided:

    R Function<T, R>
    Boolean Predicate<T>
    Void Consumer<T>
    T BinaryOperator<T>
    T UnaryOperator<T>

- Predicate<Integer> big = x → (x>8)
  - No constructor…
    Thus no generalization: big(n) ☹

- Lambdas use type inference
  - Converted into a SAM as matches the calling context
- Method references
  - *Employee::print*
- Default methods
  - Allows retroactive additions to an existing Interface
  - (Does not solve our problem of additions to system libraries)
- Library additions
  - E.g. Collections:
    List::foreach

```
c.forEach(Employee::printIn)
```

Java Streams:

Extending our functional Iterators:

```
myList<Employee> emps = new myList<>;
Functor<Employee> max = new Functor<>()
  { … }
emps.compute(max);
```

```
emps.compute(e → e.print() );       // list all employees
```

```
emps.compute(Employee::print);   // list all employees
```

```
emps   .filter( salary(1200) )
       .compute(Employee::print);        // list all rich employees
```

```
emps   .filter( salary(1200) )
       .filter( degree("CS") )
       .compute(Employee::print);        // list all rich employees
```

```
emps   .filter( salary(1200).and(degree("CS")  ))
       .compute(Employee::print);        // list all rich employees
```