# (JSR-335)

©John Kostaras

JCrete 19 – 24 August 2013

# Agenda

- What is λ
- λ in C
- λ in Ruby
- λ in Erlang
- λ in Scala
- λ in Clojure
- λ in C#
- λ in Java
- Exercises

# Have you ever come across this?

* Find a product by id:

```
public boolean find(String productId) {
   boolean found = false;
   for (Product product : products) {
      if (product.getId().equals(productId)) {
            found = true;
            break;
      }
   }
   return found;
}
```

# Have you ever come across this?

* Delete a product:

```
public Product delete(String productId) {
   Product toReturn = null;
   for (Product product : products) {
       if (product.getId().equals(productId)) {
             toReturn = product;
             products.remove(product);
             break;
       }
   }
   return toReturn;
}
```

# What is λ

- A λ (lambda) expression is an anonymous function that can be passed as argument or returned as the value of function calls.
- In the Java context they are similar to anonymous methods. Like a method  it provides a list of formal parameters and a body—an expression or block—expressed in terms of those parameters
- They make it easier to distribute processing of collections over multiple threads
- Collection methods take a function and apply it to every element

# λ advantages

- More concise code
- Ability to modify methods by supplying additional functionality
- Better support for multi-core processing

# Functional Programming

- Declarative style (not imperative)
- Immutability (avoid mutation/side-effects in most/all cases, assignments, imperative control structures)
- Use functions as first-class values or as parameters to higher-order functions
- Prefer expressions over statements (substitution model)
- A programming style that encourages recursion and recursive data structures (tail-call recursions)
- Syntax or style that is closer to traditional mathematical definitions of functions
- Using certain programming idioms related to laziness (vs. eager evaluation)

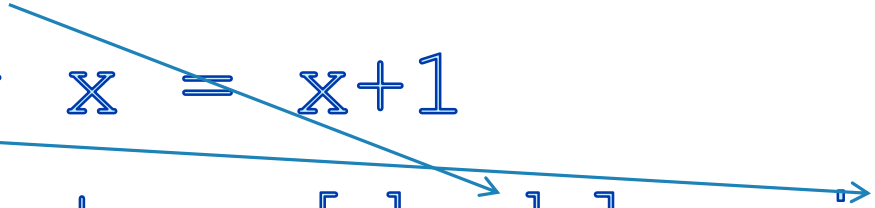
# Expressions

* An expression can be:
  * An identifier (e.g. x)
  * A literal (e.g. 5, 3.14, ”abc”)
  * A function application (e.g. Math.sqrt(x))
  * An operator application (e.g. –x, x / y)
  * A conditional expression (e.g. if (x < 0) return –x; else return x;)
  * A block (e.g. { int x = Math.abs(y) ; x *= 2; })
  * An anonymous function (e.g. x -> x + 1)

# First-class functions

* Can be used wherever we use values, e.g.

```
fun double x = 2*x
fun incr x = x+1
val a_list = [double, incr,
  double(incr 7)]
```



# Function closures

* Functions that access 'outside' variables that are in the enclosing method's scope , e.g.

```
a = 1
fun f x = x + a
val x = 2
val y = 3
val z = f(x+y)      value of z???
```

# Higher-order Functions

* Functions that take other functions as arguments or return them as results
  * pass functions to functions
  * create functions within functions
  * return functions from functions

```
fun n_times (f,n,x) =
 if n=0 then x
 else f (n_times(f,n-1,x))

fun double x = x+x
val x1 = n_times(double,4,7)          or
val x1 = n_times((fn x=>x+x),4,7)
```

# Return functions

* Functions can also return functions:

```
fun even_or_odd (n) =
    if (n mod 2 = 0)
    then fn x => 2*x
    else fn x => 3*x
```

# Functional vs OO Programming

## Functional

* functions
* bindings (closures)

## Object-oriented

* methods
* private fields & private methods

# λ in C

# λ in C (1/3)

* In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. E.g.

```
void qsort(void *list[],
           int left, int right,
           int (*comp)(void *, void *));
```

* Any pointer can be cast to `void *` and back again without loss of information, so we can call `qsort` by casting arguments to `void *`.

# λ in C (2/3)

* The fourth parameter of `qsort` says that `comp` is a pointer to a function that has two `void *` arguments and returns an int. Inside `qsort` we would have something like:

```
int result = (*comp)(list[i], list[left]);
```

* In the call to `qsort`, `strcmp` and `numcmp` are addresses of comparison functions:

```
qsort((void**) list, 0, nlines-1,
      (int (*)(void*,void*)) numcmp);
qsort((void**) list, 0, nlines-1,
      (int (*)(void*,void*)) strcmp);
```

# λ in C (3/3)

where

```
/* numcmp: compare s1 and s2 numerically */
int numcmp(char *s1, char *s2);
```

and

```
/* strcmp: compare s1 and s2 alphabetically */
int strcmp(char *s1, char *s2);
```

# λ in Ruby

# λ in Ruby (1/5)

* Blocks

```
➢ array = [1, 2, 3, 4]
➢ array.each { |i| print "#{i}" }
1234 => [1, 2, 3, 4]
➢ array.collect! { |n| n**2 }
[1, 4, 9, 16]

class Array
    def apply!
       self.each_with_index do |n, i|
          self[i] = yield(n)
       end
    end
end
➢ array.apply! { |n| n**2 }
[1, 4, 9, 16]
```

Block

# λ in Ruby (2/5)

* Procs (or Procedures) are blocks that can be saved

```
➢ square = Proc.new { |n| n**2 }
➢ array.apply!(square)
[1, 4, 9, 16]
➢ array.apply!(Proc.new { |n| n**2 })
[1, 4, 9, 16]
class Array
    def apply!(code)
        self.each_with_index do |n, i|
            self[i] = code.call(n)
        end
    end
end
```

* Blocks are also Procs

# λ in Ruby (3/5)

* Lambdas (or anonymous methods)
  - `array.apply!(lambda { |n| n**2 })`

```
[1, 4, 9, 16]
```

* lambdas check the number of arguments passed while Procs don't
* lambdas have diminutive returns, i.e. while a Proc return will stop a method and return the value provided, lambdas will return their value to the method and let the method continue

# λ in Ruby (4/5)

```
def proc_return
  Proc.new { return "Proc.new" }.call
  return "proc_return finished"
end
def lambda_return
 lambda { return "lambda" }.call
 return "lambda_return finished"
end
puts proc_return # => Proc.new
puts lambda_return # => lambda_return finished
```

# λ in Ruby (5/5)

* Method objects (named methods that allow to save lambdas)
  - ```
    def square(n)
        n**2
    end
    ```
  - ```
    array.apply!(method(:square))
    ```

```
[1, 4, 9, 16]
```

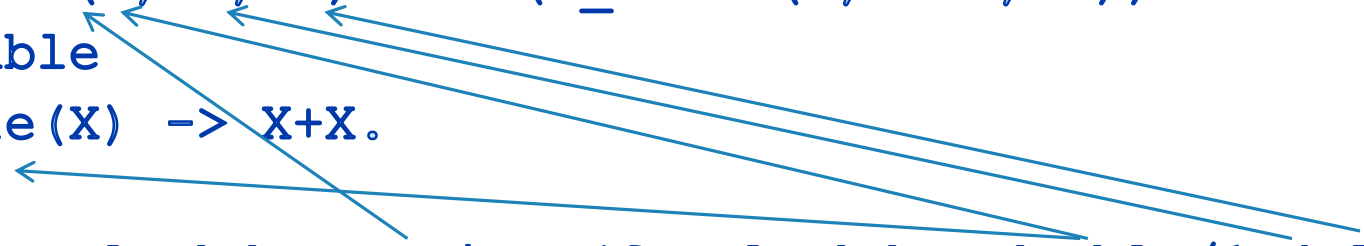
# λ in Erlang

# λ in Erlang (1/3)

* *Higher order function* is a function that can accept other functions as arguments

```
-module(lambdas).
-compile(export_all).
% n_times
n_times(F, 0, X) -> X;
n_times(F, N, X) -> F(n_times(F, N-1, X)).
% double
double(X) -> X+X.

1> X1 = lambdas:n_times(fun lambdas:double/1,4,7).
112
```



# λ in Erlang (2/3)

* Sort alphabetically and numerically

```
1> L = ["1", "9", "10", "2"].
["1","9","10","2"]
2> lists:sort(L).
["1","10","2","9"]
3> lists:sort(fun(X,Y) -> X >= Y end, L).
["9","2","10","1"]
4> lists:sort(fun(X,Y) ->
   string:to_integer(X) >= string:to_integer(Y) end,
L).
["10","9","2","1"]
```

# λ in Erlang (3/3)

* Anonymous functions

```
fun(Args1) ->
   Expression1, Exp2, ..., ExpN;
   (Args2) ->
   Expression1, Exp2, ..., ExpN;
   (Args3) ->
   Expression1, Exp2, ..., ExpN
end
```

E.g.

```
triple_n_times(N, X) -> n_times(fun (X) -> 3*X end,
N,X).
```

# λ in Scala

# Functional Support

* First class functions (or anonymous functions)

```
(x: Int, y: Int) => x * y
```

* Higher-order functions

```
def higher-order (f: Int => Int)= …
```

* Functions are treated as objects
  * `A => B` ⇔ `trait Function1[A,B] { def apply(x: A) : B }`
  * Supports traits for functions with more parameters (up to 22)

```
(x: Int) => x * x ⇔
new Function1[Int, Int]  { def  apply(x: Int) =  x * x }
```

* Tail-recursion
  * `@tailrec` annotation
* Currying
  * $def\ f(args_1)\ldots(args_{n-1})(args_n) = E \Leftrightarrow$
    $def\ f = (args_1 => (args_2 => \ldots(args_n => E)\ldots))$

```
def  sum(f: Int  => Int)(a: Int , b: Int ): Int  = if
(a > b) 0 else  f(a) + sum(f)(a + 1, b)
```

# Functional Support (cont.)

* Lists
  * map, flatMap:

```
def squareList(xs: List[Int]) =
    xs map (x => 2 * x)
```

  * filter, filterNot, partition, takeWhile, dropWhile, span:

```
def positiveList(xs: List[Int]) =
    xs filter (x => x > 0)
```

  * reduceLeft, foldLeft, reduceRight, foldRight:

```
def sum(xs: List[Int]) =
    (0 :: xs) reduceLeft (_ + _)
```

* Arrays, Sets, Maps, Streams and Strings support them, too

# Functional Support (cont.)

* Lists, Arrays, Sets, Strings, Streams:
  * `for` is a combination of higher-order functions `map`, `flatMap` and `filter`:

```
for (b <- books if b.title indexOf "Java" >= 0)
yield b.title
```

* Maps can be used anywhere functions can
* Streams are similar to lists, but their tail is evaluated only on demand (lazy evaluation)

```
((1 to 1000).toStream filter isPrime)
lazy val x = expr
```

* Infinite Streams

```
def sieve(s: Stream[Int]): Stream[Int] =
 s.head #:: sieve(s.tail filter (_  % s.head != 0))
```

# λ in C#

# λ in C# (1/8)

* λ expression:

```
x => x * x
```

* delegate type:

```
delegate int del(int i);
del myDelegate = x => x * x;
int j = myDelegate(5); //j = 25
```

* expression tree type:

```
Expression<del> myET = x => x * x;
```

# λ in C# (2/8)

* Expression lambdas:

```
(input parameters) => expression
```

E.g.

```
x => x * x
(x, y) => x == y
(int x, string s) => s.Length > x
() => SomeMethod()
```

# λ in C# (3/8)

* Statement lambdas:

```
(input parameters) => {statement;}
```

E.g.

```
c => { int l=c.Length(); return l; }
```

# λ in C# (4/8)

* Async lambdas:

async, await

E.g.

```
button1.Click += async (sender, e) => {
 await ExampleMethodAsync();
};
```

# λ in C# (5/8)

* Lambdas with the Standard Query Operators

Func

E.g.

```
// 1 input param (int), return type (bool)
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4);
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int evenNumbers = numbers.Count(n => n % 2 == 0);
var firstNumLessThan6=numbers.TakeWhile(n => n<6);
```

# λ in C# (6/8)

* Type inference in Lambdas

E.g.

```
customers.Where(c => c.City == "Chania");
```

* The lambda must contain the same number of parameters as the delegate type.
* Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
* The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

# λ in C# (7/8)

* Variable scope in Lambda expressions
  * Lambdas can refer to outer variables (aka closures) that are in scope in the enclosing method or type in which the lambda is defined.
  * An outer variable must be definitely assigned before it can be consumed in a lambda expression.

E.g.

```
delegate bool D();
delegate bool D2(int i);
public void aMethod (int input) {
int j = 0;
// access to 2 outer vars
del = () => { j = 10; return j > input; };
// lives even when aMethod() goes out of scope
del2 = (x) => {return x == j; };
```

# λ in C# (8/8)

* Variable scope in Lambda expressions
  * A variable that is captured will not be garbage-collected until the delegate that references it goes out of scope.
  * Variables introduced within a lambda expression are not visible in the outer method.
  * A lambda expression cannot directly capture a ref or out parameter from an enclosing method.
  * A return statement in a lambda expression does not cause the enclosing method to return.
  * A lambda expression cannot contain a goto statement, break statement, or continue statement whose target is outside the body or in the body of a contained anonymous function.

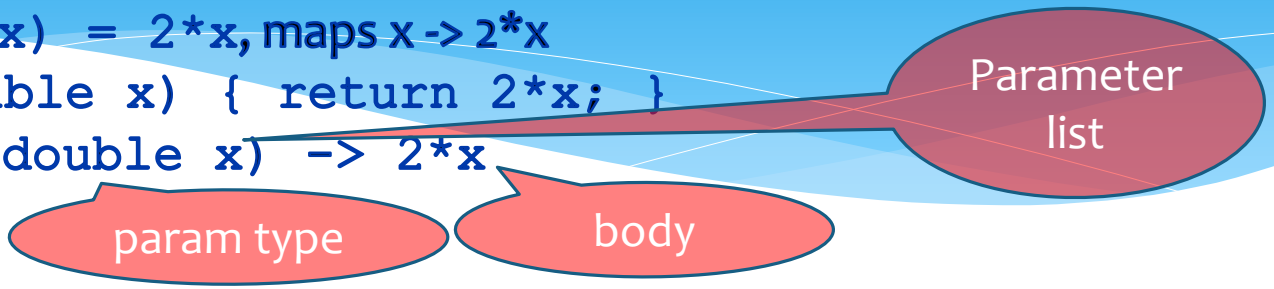
# λ in Java

# Disclaimer

At the time of this writing, neither the JSR-335 nor the Java API for lambdas have been finalized. Some of the syntax found in these slides might be different than in the final version of the API.

# λ-expressions

* f(x) = y, e.g. f(x) = 2*x, maps x -> 2*x
* double dbl(double x) { return 2*x; }
  ➔ (double x) -> 2*x




```
FileFilter directoryFilter = new FileFilter() {
  public boolean accept(File file) {
    return file.isDirectory();
  }
};
FileFilter directoryFilter =
    (File f) -> f.isDirectory();
// f -> f.isDirectory();
```

* Lambdas are lexically scoped, meaning that a lambda recognizes the immediate environment around its definition as the next outermost scope.

# λ expressions – Exercise 1

```
JButton button = new JButton("Click me!");
button.addActionListener(
 new ActionListener() {
   public void actionPerformed(ActionEvent e)
{
JOptionPane.showMessageDialog(frame, "Button
clicked!");
    }
});
```

IDEA: Settings | IDE Settings | Editor | Code Folding | "Closures"

# Syntax

Syntax:

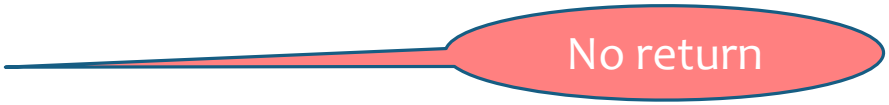
```
 (parameters) -> expression
```

or

```
 (parameters) -> { statements; }
```

E.g.

```
(int x, int y) -> x + y
```




```
(x, y) -> x % y
() -> Math.pi
(String s) -> s.toUpperCase()   or
String::toUpperCase                  ⇐ Method Reference
x -> 2 * x
c -> { int n = c.size(); c.clear(); return n ; }
```

# Collections – common tasks

## External iteration

- Control flow external to the collection (external iterator)

## Internal iteration

- Collection manages the iterator internally

- Filter a collection *(filter)*
- Apply a transformation to each element of a collection *(map)*
- Build up an overall value on the entire collection (e.g. sum, average etc.) *(reduce)*

`java.util.functions`

- `Predicate` *(filter)*
- `Mapper` *(map)*
- `Collector` *(reduce)*

# Filter (Predicate)

```
final List<Integer> even = new ArrayList<>();
for (Integer i : list) {
    if (i % 2 == 0) {
        even.add(i);
    }
}
```

DRY

i % 2 == 0

```
Predicate<Integer> even = i -> i % 2 == 0;
list.filter(even);
// list.filter(i -> i % 2 == 0);
```

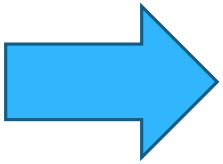
```
list.stream().filter(even).Collectors(
   Collector.toList());
```

# Map (Mapper)

```
final List<Integer> dbl = new ArrayList<>();
for (Integer i : list) {
      dbl.add(i * 2);
}
```

---

```
Mapper<Integer> dbl = i -> i * 2;
list.map(even);
// list.map(i -> i * 2);
```



---

```
Function<Integer, Integer> dblf = i -> i * 2;
        list.stream().map(dblf).
                Collectors(Collector.toList());
```

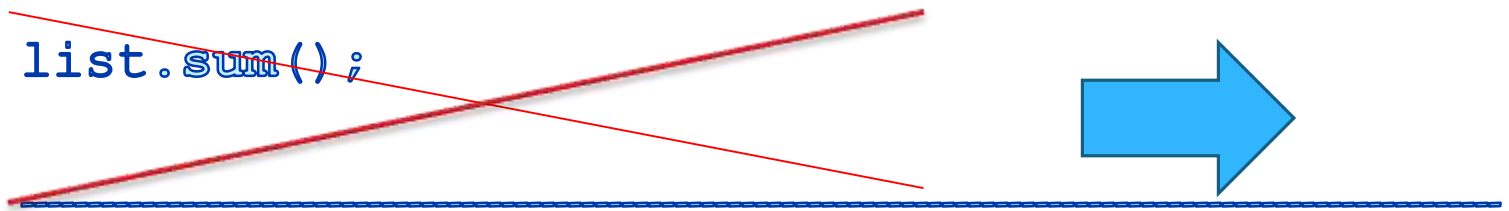
Input type

Return type

# Reduce

```
int sum;
for (Integer i : list) {
    sum += i;
}
```

```
list.sum();
```



```
list.stream().mapToInt(i -> i).sum();  // or
Optional<Integer> sum =
list.stream().reduce((x, y) -> { return x + y;} );
sum.ifPresent(s -> System.out.println("sum =", s));
```

# java.util.Stream

* Stream is like a pipeline that carries data from a source; it doesn't store any values like e.g. a Collection does
* Is functional in nature; it doesn't modify the original data
* Lazy optimisations; operations as separated to intermediate (lazy) and terminal (eager); all processing is deferred until the terminal operation starts
* Infinite streams are allowed (to complete in time)

# Stream interface

| methodT | parameter | returns | λ | lazy/eager |
|---|---|---|---|---|
| map | `Function<T,R>` | `Stream<R>` | T -> R | lazy |
| filter | `Predicate<T>` | `Stream<T>` | T -> boolean | lazy |
| sorted | `Comparator<T>` | `Stream<T>` | (T, T) -> int | lazy |
| limit, substream, skip | `int` | `Stream<T>` | | lazy |
| forEach | `Consumer<T>` | `void` | T -> void | eager |
| findFirst | `Predicate<T>` | `Optional<T>` | T -> boolean | eager |
| reduce | `BinaryOperator<T>` | `Optional<T>` | (T,T) -> T | eager |
| collect | `Supplier<R>` | `R` | R -> R | eager |
| min, max | `Comparator<T>` | `Optional<T>` | (T, T) -> int | eager |
| flatMap | `Function<T,Stream<R>>` | `Stream<R>` | T -> Stream<R> | lazy |
| sorted | `Stream<T>` | `Stream<T>` | T -> T | lazy |

# java.util.function

* Consumer<T>
* Supplier<T>
* Predicate<T>
* Function<T,R>

# java.util.function.Consumer<T>

```
// accepts a single input argument and returns
// no result; used as parameter to forEach()
interface Consumer<T> {
    void accept(T t);// T -> void
}
```

Exercise: Iterate over a list and print it to screen

# java.util.function.Supplier<T>

```
// a factory that returns either a new
// instance or a pre-created instance.
// To create lazy infinite Streams and as
// the parameter to the Optional class'
// orElse() method.
interface Supplier<T> {
    T get();    // () -> T
}
```

# java.util.function.Predicate<T>

```
// Checks if the input object matches some
// criteria. Used to store a lambda expression and as a
// parameter to Stream's methods like
// filter() and anyMatch().
interface Predicate<T> {
    boolean test(T t);    // T -> boolean
    default Predicate<T> and(Predicate<? super T>
other);
    default Predicate<T> or(Predicate<? super T> other);
    default Predicate<T> negate();
    static <T> Predicate<T> isEqual(Object other);
}
```

# java.util.function.Function<T, R>

```
// a function that transforms the input <T> to
// output <R>. Used as a parameter to Stream's
// map() method.
interface Function<T, R> {
    R apply(T t);          // T -> R
    default <V> Function<V,R> compose(Function<?
super V, ? extends T> before);
    default <V> Function<T,V> andThen(Function<?
super R, ? extends V> after);
    static <T> Function<T, T> identity();
}
```

# java.util.Optional<T>

* Optional<T>  // ifPresent()

E.g.

```
Optional<String> found = prices.stream().
      filter(name -> name.startsWith(letter)).
      findFirst();
found.orElse("Not found!");
found.ifPresent(name -> name.append("!"));
```

# Default methods

* In addition to abstract methods, interfaces can have methods with implementation, marked as default since Java 8. These methods are automatically inherited by classes that implement the interfaces. This way existing classes are enhanced with new methods without having to change.
* Subclasses automatically carry over the default methods from their superclasses;
* implementation in a subclass takes precedence over the one in superclass;
* Implementations in classes take precedence over all interface defaults;
* the inheriting class should disambiguate conflicts

# Collections.forEach

```
public class Collections {
  public void forEach(
      Consumer<? super T> consumer);
}
interface Consumer<T> { void accept(T t); }
pointList.forEach(new Consumer() {
        public void accept(Point p) {
            p.move(p.y, p.x);
        }
    });
```

Higher order function

---

```
pointList.forEach(p -> p.move(p.y, p.x));
```

# First-class functions

* Can be used wherever we use values, e.g.

```
Function<Person,String> byName
  = person -> person.getName();
```

# Function closures

* Functions that use variables defined outside of them, e.g.

```
public static Predicate<String>
  startsWith(final String letter)
  {
     return name ->
  name.startsWith(letter);
}
```

`letter` is defined in the method that contains the `startsWith()` function

and `letter` needs to be "effectively final"

# Higher-order Functions

* Functions that take or return other functions
  * pass functions to functions
  * create functions within functions
  * return functions from functions

```
prices.stream().mapToDouble(
     price -> price * 1.23).sum();
public static Predicate<String>
  startsWith(final String letter) {
    return name -> name.startsWith(letter);
}
```

# Return functions

* Higher-order functions can also return functions:

```
Function<Integer, Integer>
evenOrOdd(int n) {
   if (n % 2 == 0)
      return x -> 2*x;
   else return x -> 3*x;   }
```

# Strings

```
s.chars().forEach(ch ->
      System.out.println(ch));
s.chars().forEach(System.out::println);
s.chars().forEach(IterateString::printChar);
s.chars().map(ch ->
      Character.valueOf((char)ch))
.forEach(System.out::println);
s.chars().filter(ch -> Character.isDigit(ch))
.forEach(ch -> printChar(ch));
s.chars().filter(Character::isDigit)
.forEach(IterateString::printChar);
```

**Output: ASCII**

# Comparators

```
interface Comparator {
      int compare(Object o1, Object o2);
}
list.stream().sorted((Person p1, Person p2)
      -> (p1.compareTo(p1)).collect(toList());
list.stream().sorted(Person::compareTo(p1)
                .collect(toList());
final Function<Person, String> byName =
                     person -> person.getName();
people.stream().sorted(comparing(byName));
```

Comparators

# reduce

```
<I> I reduce(I identity,
             BiFunction<I, ? super T, I> accumulator,
             BinaryOperator<I> combiner);
E.g.
books.stream().reduce(0,
                      (sum, b) -> sum + book.getPrice(),
                      Integer::sum);
```

# Collector

```
<R> R collect(Supplier<R> resultFactory,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R> combiner);
```

A collect operation requires three things:

- ✓ a factory function which will construct new instances of the result container
- ✓ an accumulating function that will update a result container by incorporating a new element, and
- ✓ a combining function that can take two result containers and merge their contents

E.g.

```
books.stream().collect(ArrayList::new,
                       (c, b) -> c.add(book.getPrice()),
                       ArrayList::addAll);
```

# Collector (cont.)

E.g.

```
books.stream().
    collect(Collectors.groupingBy(Book::getAuthor));
```

returns a Collector for grouping sets of elements based on some key

# List files

```
Files.list(Paths.get("."))
.filter(Files::isDirectory)
.forEach(System.out::println);

Files.newDirectoryStream(Paths.get("."),
  path -> path.toString().endsWith(".java"))
.forEach(System.out::println);
```

# λ in Java - Functional interfaces

```
interface Runnable { void run(); }
interface Callable<V> {
   V call() throws Exception; }
interface ActionListener {
   void actionPerformed(ActionEvent e); }
interface Comparator<T> {
   int compare(T o1, T o2);
   boolean equals(Object obj); }
-----------------------------------------
Runnable r = () -> {};  // λ expression
Callable<Runnable> c =
  () ->  () -> { System.out.println("hi"); };
Callable<Integer> c = flag ?
  (() -> 23) : (() -> 42);
```

# Recursion

* λ-expressions can be used to define recursive functions provided that the recursive call uses a name defined in the enclosing environment of the λ. E.g.

```
UnaryOperator<Integer> factorial = i ->
    { return i == 0 ? 1 : i * factorial.apply( i - 1 ); };
```

  * factorial must be declared as an instance or static variable
* If a function calls itself as its last action, the function's stack frame can be reused. This is called *tail recursion*.
* If the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called *tail-calls*.
* E.g.

```
int factorial(int n) {
if  (n == 0) return 1;
else return n * factorial(n - 1); }
```

* ***Memoization*** is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously processed inputs.

# Lazyness

* Eager (simplicity) vs. lazy (efficiency).
* Delayed initialisation:

```
class Supplier<T> { T get(); }
Supplier<Heavy> supplier = () -> new Heavy();
Supplier<Heavy> supplier = Heavy::new;
```

* Streams have two types of methods: *intermediate* and *terminal* operations. We chain multiple *intermediate* operations (e.g. `filter`, `map`) followed by a *terminal* operation (e.g. `sum`, `reduce`, `findFirst`). Only when one of the terminal operations is called, are the cached core behaviors executed.

# Lazyness (cont.)

* E.g.

```
final String firstJavaFileName =
filenames.stream()
.filter(name -> name.endsWith(".java"))
.map(name -> name.toUpperCase())
.findFirst()
.get();
```

Evaluation starts from right to left; `filter()` finds the first filename that satisfies the condition and returns it to mapper.

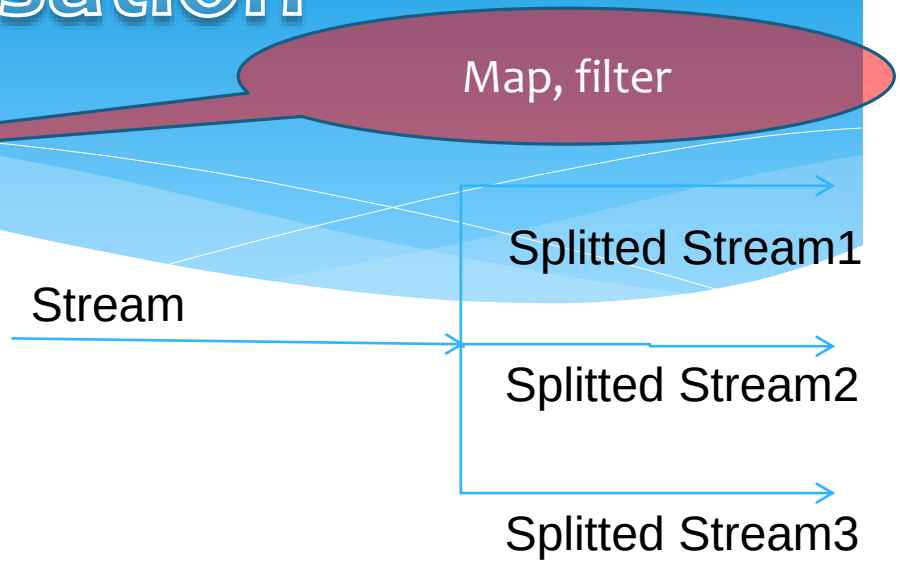
# Lazyness (cont.)

* Infinite streams

```
Streams.iterate( seed, UnaryOperator )
.limit( count )
.collect(Collectors.toList());
```

# Parallelisation

```
list.parallelStream(). …
* java.util.Spliterator
   * tryAdvance()   - iterate
   * trySplit()     - split
```




1. Do we really want to run the lambda expressions concurrently?
2. Can the code run independently without causing any side-effect or race conditions?
3. The correctness of the solution should not depend on the order of execution of the lambda expressions that are scheduled to run concurrently.

# Functional interfaces

```
@FunctionalInterface
public interface MyInterface<T> {
    MyInterface<T> apply();
    default boolean isComplete() {
        return false;
    }
    //...
}
```

A *functional interface* is an interface that has just one abstract method (and zero or more default or implemented methods), and thus represents a functional contract.

# Recap - Syntax

* Functional interfaces: `@FunctionalInterface`
* No parameter λ-expressions:

```
() -> System.out.println();
```

* Single parameter λ-expression:

```
(String name) -> System.out.println(name);
(name) -> System.out.println(name);
name -> System.out.println(name);
```

* Multi-parameters λ-expression:

```
(name1, name2) -> name1.equals(name2);
```

# Recap – Syntax (cont.)

* Mixed-parameters λ-expression:

```
(false, (name1, name2)) ->
name1.equals(name2);
```

* Store a λ-expression:

```
Predicate<String> endsWithJava =
      name -> name.endsWith(".java");
```

* Return a λ-expression (and closures):

```
public static Predicate<String>
checkIfEndsWith(final String pattern) {
   return name -> name.endsWith(pattern);
}
```

# Recap – Syntax (cont.)

* Return a λ-expression from a λ-expression:

```
Function<String, Predicate<String>>
checkIfEndsWith =
pattern -> name -> name.endsWith(pattern);
```

* Method reference of an instance method:

```
name::toLowerCase
[<==> name -> name.toLowerCase()]
```

* Method reference of a static method:

```
Character::isDigit
[<==> c -> Character.isDigit(c)]
```

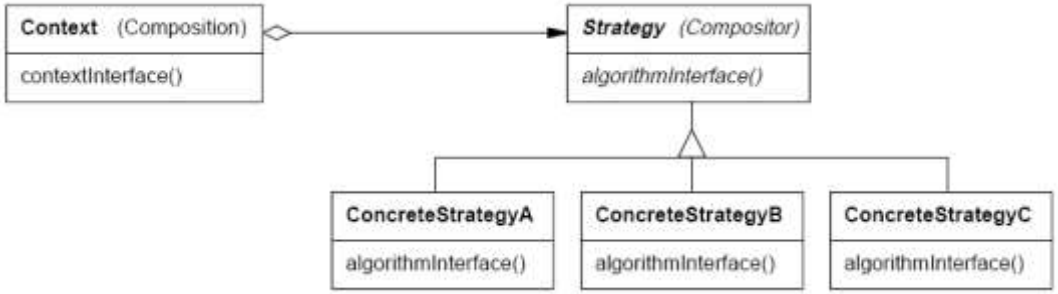
* Constructor reference:

```
MyClass::new
[<==> () -> new MyClass()]
```

# Strategy pattern

* See strategy example
* Converts http://javacamp.org/designPattern/strategy.html to lambdas
* See http://www.javacodegeeks.com/2013/07/strategy-pattern-using-lambda-expressions-in-java-8.html too

```
public interface FortuneCookies {  void print(); }
FortuneCookies fc = () -> System.out.println("You got nothing");
fc.print();
```



# Resources

* JSR-335 http://www.jcp.org/en/jsr/detail?id=335
* http://en.wikipedia.org/wiki/Closure_(computer_science)
* "Maurice Naftalin's Lambda FAQ", http://www.lambdafaq.org/.
* https://github.com/AdoptOpenJDK/lambda-tutorial
* http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
* http://www.functionaljava.org
* Neward T. (2013), „Java 8: Lambdas – Part 1”, *Java Magazine*, Issue 13, July-August, pp. 34-40.
* http://openjdk.java.net/projects/lambda/
* http://jdk8.java.net

# Resources (cont.)


* Kerninghan B.W. & Ritchie D.M. (1988), *The C Programming Language*, 2nd Edition, Prentice Hall.
* "Lambda Expressions (C# Programming Guide)", http://msdn.microsoft.com/en-us/library/bb397687.aspx.
* Sosinski R. "Understanding Ruby Blocks, Procs and Lambdas", http://www.robertsosinski.com/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/.
* LearnYouSomeErlang, "Higher order functions", http://learnyousomeerlang.com/higher-order-functions.
* Grossman D. (2013), *Programming Languages*, University of Washington, http://www.coursera.org.
* Odersky M. (2012), *Functional Programming Principles in Scala*, EPFL, http://www.coursera.org.
* https://en.wikipedia.org/wiki/Tail_call
* http://en.wikipedia.org/wiki/Memoization

# Recap & Questions

* Java 8 lambdas is *not* an implementation for full functional support
* How do I define a Function<T,R> that takes >1 arguments?
  * Use `@FunctionalInterface`
* What about **currying**?
* What about combining functions – h(g(f(x))) – in Java?

# Questions