FUNCTIONAL PROGRAMMING GOES MAINSTREAM

Carlos Muñiz Solaz

ESAC

27 June 2018

AGENDA

- FUNCTIONAL PROGRAMMING CONCEPTS
- FUNCTIONAL FEATURES IN PYTHON
- FUNCTIONAL FEATURES IN JAVA
- CONCLUSIONS

PROBLEMS WITH TODAY'S PROGRAMMING

- **IMPERATIVE PROGRAMMING** FOCUSES ON TELLING A COMPUTER **HOW** TO DO THINGS

- **IMPERATIVE PROGRAMMING** IS A **PROGRAMMING PARADIGM** THAT USES STATEMENTS THAT CHANGE A **PROGRAM'S STATE**

- **OBJECT ORIENTED PROGRAMMING** IS A PROGRAMMING PARADIGM THAT MODELS INTERACTION WITH REAL WORLD (OBJECTS, CLASSES AND METHODS)

IMPERATIVE PROGRAMMING: EXAMPLE

```
int[][] rows = new int[][] {{1,2,3,4},{5,8,7,2},{2,1,6,5}};

int total = 0;
for(int i=0; i<rows.length; i++) {
    int rowSum = 0;
    for(j=0; j<rows[i].length; j++) {
        rowSum += rows[i][j];
    }
    total += rowSum * rowSum;
}
```

WHAT IS MAKING OUR CODE MESSY ?

- WE WRITE TOO MUCH OF **ARTIFACT** CODE
- WE USE **MUTABLE STATES**
- WE USE **SHARED MUTABLE STATES**

CAN WE DO BETTER ?

- **FUNCTIONAL PROGRAMMING** ALLOWS US TO TELL THE COMPUTER **WHAT** WE WANT TO DO

- **FUNCTIONAL PROGRAMMING** IS A **DECLARATIVE** PARADIGM, WHICH MEANS PROGRAMMING IS DONE WITH **EXPRESSIONS** OR **DECLARATIONS** INSTEAD OF STATEMENTS

- FUNCTIONAL PROGRAMMING TREATS COMPUTATION AS THE EVALUATION OF **MATHEMATICAL FUNCTIONS** AND AVOIDS **CHANGING-STATE** AND **MUTABLE DATA**

"Programs must be written for people to read, and only incidentally for machines to execute."

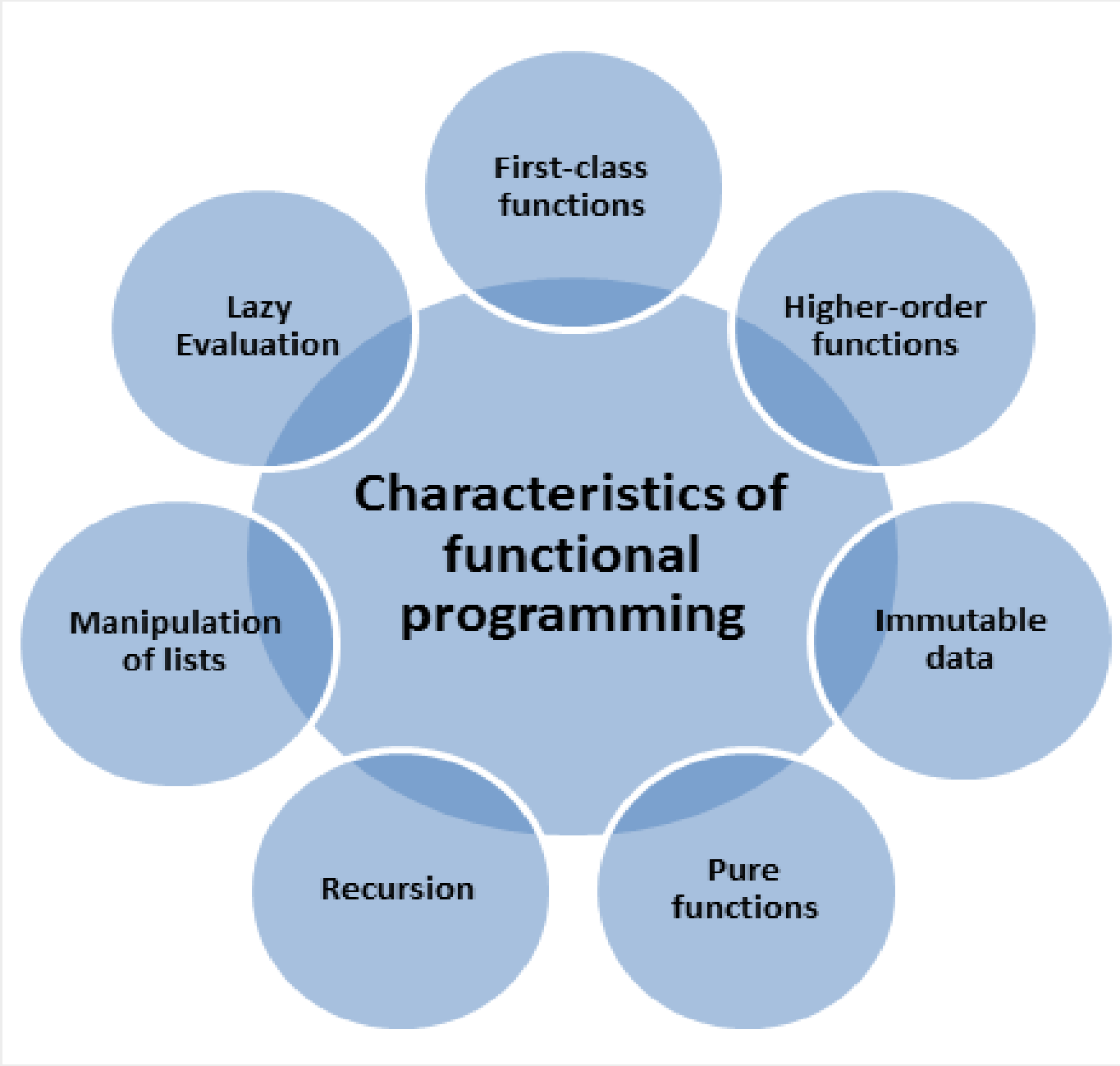
Harold Abelson, SICP

FUNCTIONAL PROGRAMMING: EXAMPLE

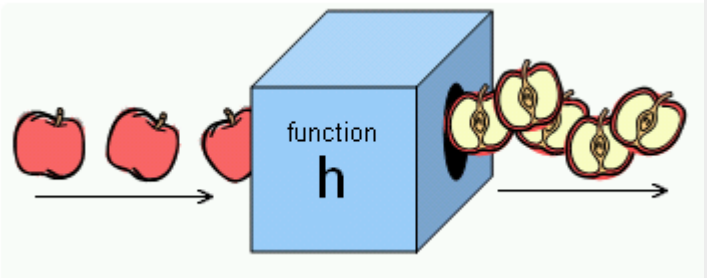
```
rows = [[1,2,3,4],[5,8,7,2],[2,1,6,5]]
square x = x * x
total = sum (map square (map sum rows))
```

FUNCTIONAL PARADIGM

- Set of ideas, not a set of strict guidelines
- Deals with calculations (algebra)
- Uses **functions** to do that
- **Mimize** the use of **variables**
- **Tries to minimize conditional statements**
- **No loops**


Characteristics of functional programming
First-class functions
Higher-order functions
Immutable data
Pure functions
Recursion
Manipulation of lists
Lazy Evaluation

PURE FUNCTIONS



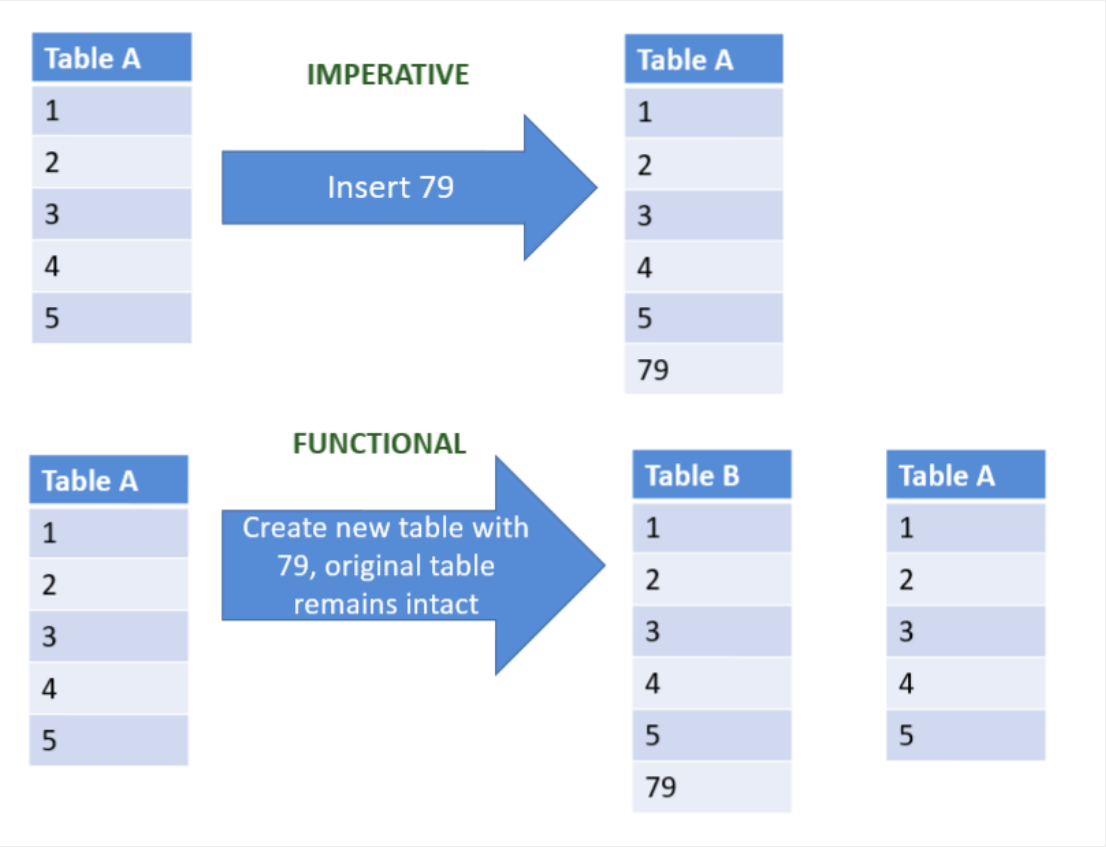
- The result depends only on input values, not state
- Evaluation does not cause side-effects

```
function priceAfterTax(productPrice) {
 return (productPrice * 0.20) + productPrice;
}
```

Advantages:

- Every time you call the function with the same input, you get the same output
- The order of functions can be changed
- Referential Transparency

IMMUTABLE DATA

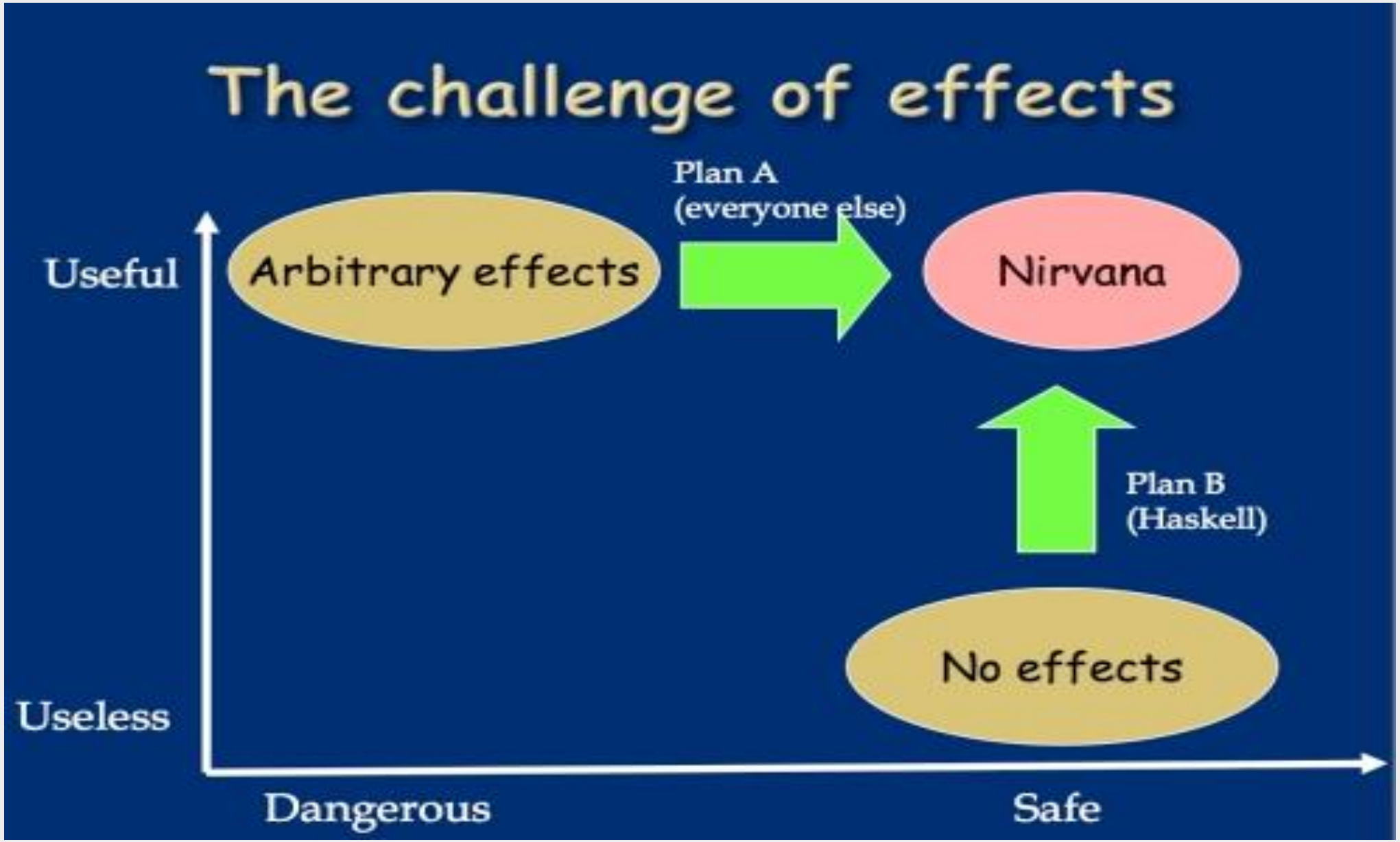


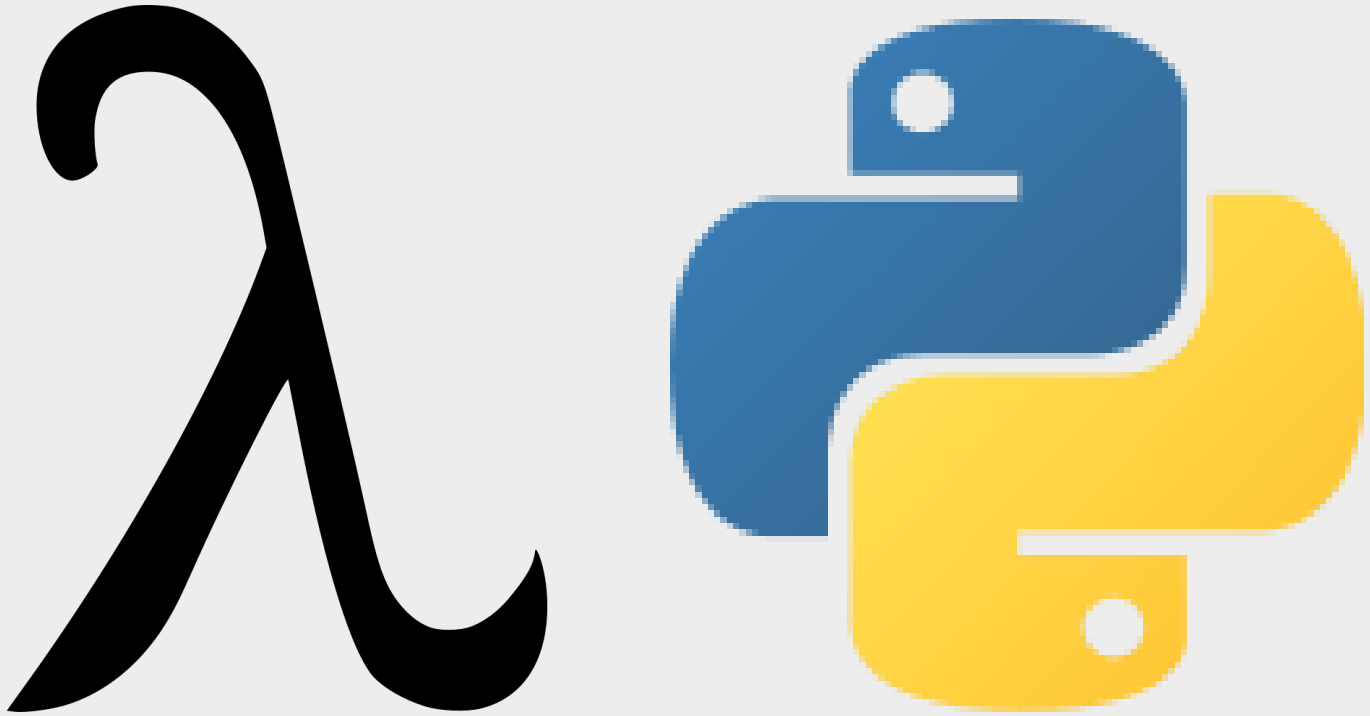
HIGH-ORDER FUNCTIONS

Higher-order functions are functions that accept other functions as argument or return them as result

The three classics:

- **map**
  Apply a function to each element of a list
- **reduce**
  Reduce a list to a single value by successively applying a binary operation
- **filter**
  Remove elements from a list


The challenge of effects
Plan A
(everyone else)
Useful
Arbitrary effects
Nirvana
Plan B
(Haskell)
No effects
Useless
Dangerous
Safe



FUNCTIONAL FEATURES

- LAMBDA FUNCTIONS
- LIST COMPREHENSIONS
- DECORATORS
- ELIMINATING LOOPS: MAP, REDUCE AND FILTER

LAMBDA FUNCTIONS

Lambda function is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created

```
is_vowel = lambda c: c.lower() in "aeiou"
is_cons = lambda c: c.lower() in "bcdfghjklmnpqrstvwxyz"
```

```
>>> is_vowel('c')
False
```

Lambda functions are mainly used in combination with the functions **filter()**, **map()** and **reduce()**.

LIST COMPREHENSIONS

We usually write:

```
collection = list()
for datum in data_set:
    if condition(datum):
        collection.append(datum)
    else:
        new = modify(datum)
        collection.append(new)
```

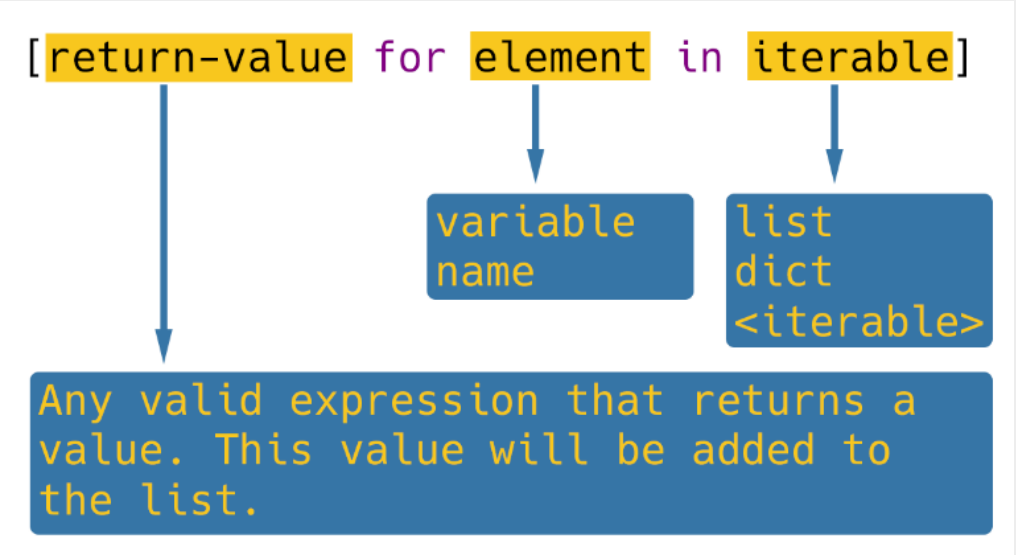
LIST COMPREHENSIONS (FUNCTIONAL WAY)

```
collection = [d if condition(d) else modify(d)
              for d in data_set]
```

- Declarative form
- More compact
- Focus from the “how” to the “what”

LIST COMPREHENSIONS (ANOTHER EXAMPLE)

In general:




```
numbers = [1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
evens = [n for n in numbers if n % 2 == 0]
```

DECORATORS

A **decorator** wraps a function, modifying its behavior.

```
def timing_function(some_function):

    """
    Outputs the time a function takes
    to execute.
    """

    def wrapper():
        t1 = time.time()
        some_function()
        t2 = time.time()
        return "Time it took to run the function: " + str((t2 - t1)) + "\n"
    return wrapper


def my_function():
    num_list = []
    for num in (range(0, 10000)):
        num_list.append(num)
    print("\nSum of all the numbers: " + str((sum(num_list))))


my_function()
```

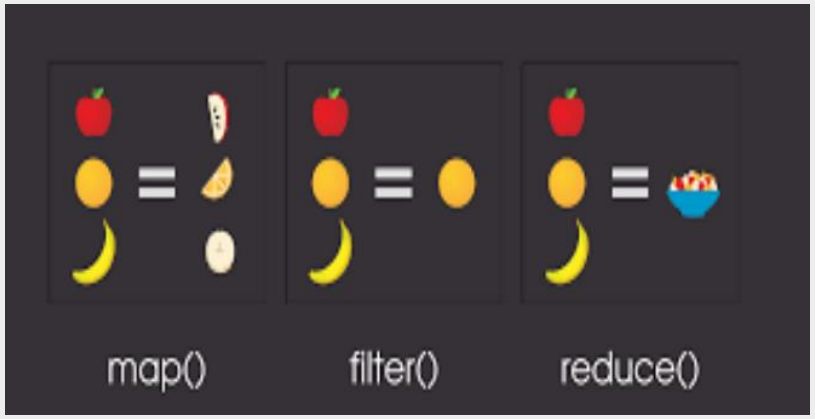
ELIMINATING LOOPS

In functional programing, there are two ways of eliminating loops:

- Recursion

```
def recur_factorial(n):
    """Function to return the factorial
    of a number using recursion"""
    if n == 1:
        return n
    else:
        return n*recur_factorial(n-1)
```

- High-order functions



ELIMINATING LOOPS: MAP

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection.

```
def multiply2(a):
    return (a*2)

test_list = [3,5,6,7,8,9,11,12]

# Multiply every element of list by 2
return_list = list(map(multiply2, test_list))
```

ELIMINATING LOOPS: FILTER

Filter takes a function and a collection of items. It filters out all the elements of the collection for which the function returns true.

```
def greater_elem(a):
    if a > 10:
        return True
    else:
        return False

# Filter if elements are greater than 10
return_list = list(filter(greater_elem, test_list))
```

In this example we call the *greater_elem*() function on every element of the list and return the elements which are True for the function.

ELIMINATING LOOPS: REDUCE

Reduce takes a function and a sequence and applies the function continually on the sequence and returns a single value.

```
reduce(lambda x,y: x*y, [47,11,42,13])
```

In this example, we are calculating the product of all elements in the list. So the evaluation order works like:

```
(((47 * 11) * 42) * 13)
```


Java™
8
(x) -> x + x

FUNCTIONAL FEATURES

- LAMBDA EXPRESSIONS
- FUNCTIONAL INTERFACES
- DEFAULT METHODS
- STREAMS

LAMBDA EXPRESSIONS: EXAMPLE

Lambda is an anonymous function. Lambda expressions are just like functions and they accept parameters just like functions.

```java
// A Java program to demonstrate simple lambda expressions
import java.util.ArrayList;
class Test
{
    public static void main(String args[])
    {
        // Creating an ArrayList with elements
        // {1, 2, 3, 4}
        ArrayList<Integer> arrL = new ArrayList<Integer>();
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);

        // Using lambda expression to print all elements
        // of arrL
        arrL.forEach(n -> System.out.println(n));

        // Using lambda expression to print even elements
        // of arrL
        arrL.forEach(n -> { if (n%2 == 0) System.out.println(n); });
    }
}
```

Output :

```
1
2
3
4
2
4
```

FUNCTIONAL INTERFACES

- An interface with **single** abstract method is called **functional interface.** For e.g: java.lang.Runnable

- A functional interface can have any number of default methods

- Lambda expressions implement the only abstract function and therefore implement functional interfaces

FUNCTIONAL INTERFACES: EXAMPLE

```
class Test
{
    public static void main(String args[])
    {
        // create anonymous inner class object
        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println("New thread created");
            }
        }).start();
    }
}
```

Output:

```
New thread created
```

```
class Test
{
  public static void main(String args[])
  {

    // lambda expression to create the object
    new Thread(()->
       {System.out.println("New thread created");}).start();
  }
}
```

Output:

```
New thread created
```

FUNCTIONALINTERFACE ANNOTATION

@FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method. In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message. However, it is not mandatory to use this annotation.

```
@FunctionalInterface
interface Square
{
    int calculate(int x);
}

class Test
{
    public static void main(String args[])
    {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x)->x*x;

        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

BUILT-IN FUNCTIONAL INTERFACES

Java 8 contains many built-in functional interfaces like:

```
Runnable r = () -> {}
```

```
Consumer c = (input) -> {}
```

```
Supplier s = () -> {output}
```

```
Function f = (input) -> {output}
```

```
BiConsumer bc = (input1, input2) -> {}
```

```
UnaryOperator negate = integer -> -integer
```

```
BinaryOperator add = (int1, int2) -> int1 + int2
```

```
Predicate p = input -> boolean
```

```
BiPredicate bp = (input1, input2) -> boolean
```

BUILT-IN FUNCTIONAL INTERFACES: EXAMPLE

```
public static void main(String args[])
{

    // create a list of strings
    List<String> names =
        Arrays.asList("Geek","GeeksQuiz","gl","QA","Geek2");

    // declare the predicate type as string and use
    // lambda expression to create object
    Predicate<String> p = (s)->s.startsWith("G");

    // Iterate through the list
    for (String st:names)
    {
        // call the test method
        if (p.test(st))
            System.out.println(st);
    }
}
```

Output:

```
Geek
GeeksQuiz
Geek2
```

DEFAULT METHODS

Before Java 8, interfaces could have only abstract methods.

Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

```
interface TestInterface
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
      System.out.println("Default Method Executed");
    }
}
```

Allow Oracle to extend collection Interfaces

```
Iterable.forEach(Consumer<? super T> action)
```

STREAMS

- Collections are for storing
- Streams are for operations

Most often, it is required to process operations rather than to store data

STREAMS: EXAMPLE

- map
- filter
- distinct
- skip, limit
- peek

- min, max
- count
- findAny, findFirst

```
users.stream()
     .map(...)
     .filter(...)
     .distinct()
     .skip(5)
     .limit(10)
     .peek(item -> System.out.print(item))
     .count();
```

STREAMS OPERATIONS

Intermediate Operations:

- **map:** The map method is used to map the items in the collection to other objects according to the Predicate passed as argument.

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

- **filter:** The filter method is used to select elements as per the Predicate passed as argument.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
```

- **sorted:** The sorted method is used to sort the stream.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().sorted().collect(Collectors.toList());
```

STREAMS OPERATIONS

Terminal Operations:

- **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.

```
List number = Arrays.asList(2,3,4,5,3);
Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

- **forEach:** The forEach method is used to iterate through every element of the stream.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

- **reduce:** The reduce method is used to reduce the elements of a stream to a single value.
  The reduce method takes a BinaryOperator as a parameter.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```

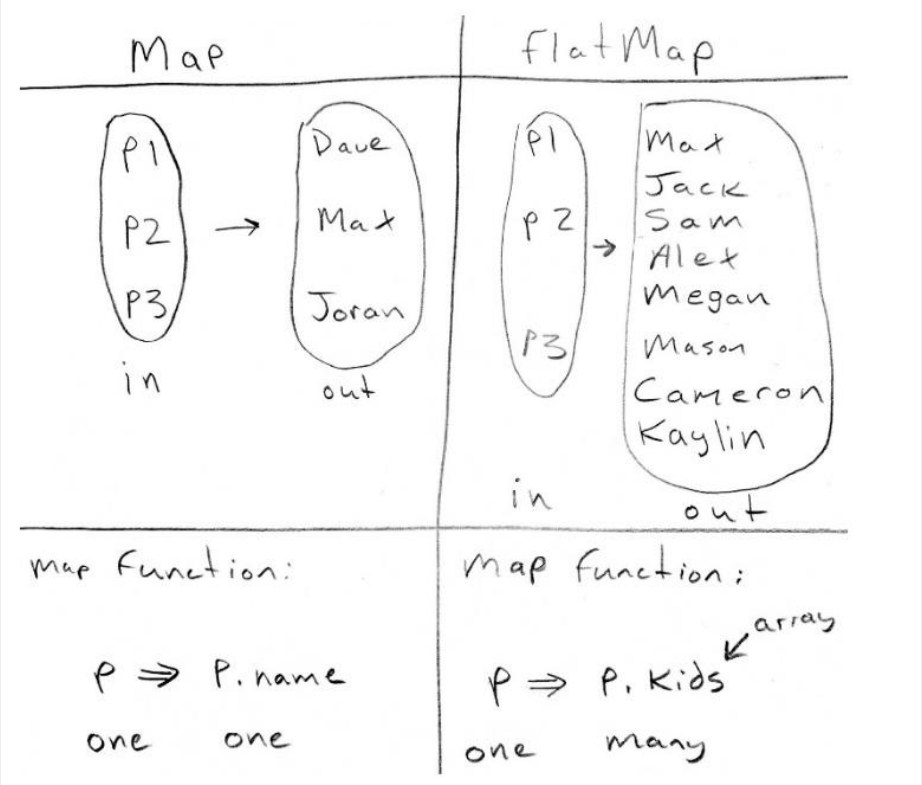
PARALLEL EXECUTION

```
List inputs = ...

inputs
    .parallelStream()
    .map(new HeavyOperation("A"))
    .map(new HeavyOperation("B"))
    .map(new HeavyOperation("C"))
    .findFirst();
```

MANY OTHER FUNCTIONAL FEATURES

- flatMap
- Optional<T>
- Pattern Matching
- String Immutability
- Final keyword



WHY GO FUNCTIONAL ?

- IT IS A NEW WAY OF THINKING
- IT IS FUN !
- IT IS MORE EXPRESSIVE
- LESS CODE, LESS BUGS, LESS ABSTRACTIONS
- CLOSER TO SPECIFICATIONS
- YOU CAN PROVE IT, INSTEAD OF TESTING
- EASIER TO PARALELLIZE

- BECAUSE EVERYBODY IS DOING IT …

BUT REMEMBER . . .

IT IS JUST ONE MORE TOOL IN OUR TOOLBOX

QUESTIONS ?