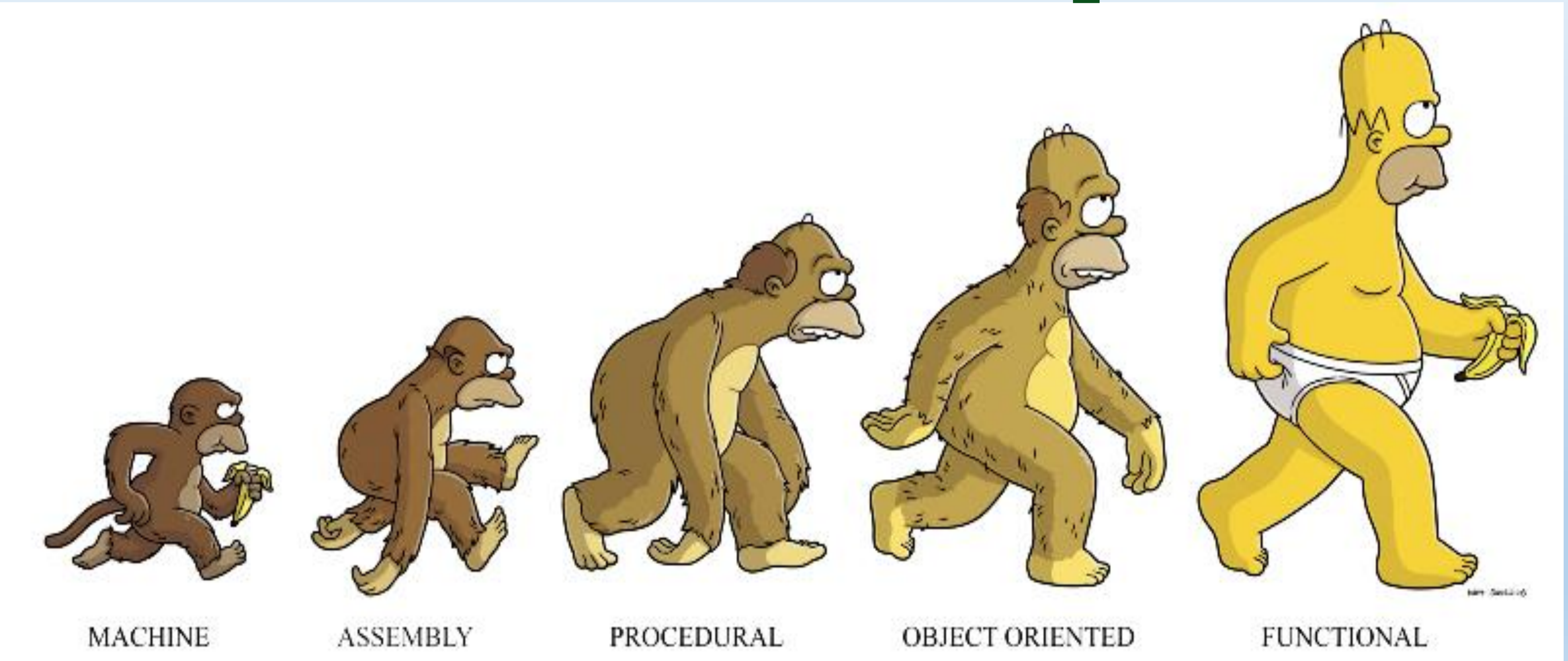
Lambda Expressions



λ

"Lambda Expression" Intro

- Introduced in Java 8
- Finally "full class functions"!
    - at least from the programmer's point of view
    - Under the covers, this is still anonymous inner classes

- (parm1,parm2) - > function of parm1 and parm2

- Defines an anonymous method
- If the lambda appears in the context of a single abstract method interface, the lambda is assumed to implement that interface's method!

Functions as a First Class Citizen

- If there was just some way of packaging the compareTo function
- And then passing that function as an argument to Arrays.sort
- Then we wouldn't need an object
  - We wouldn't need a Comparator interface
  - We wouldn't need an anonymous inner class or an explicit class
  - We wouldn't need to pass data to the Arrays.sort method

But how can we "package" a function?

Lambda Expressions

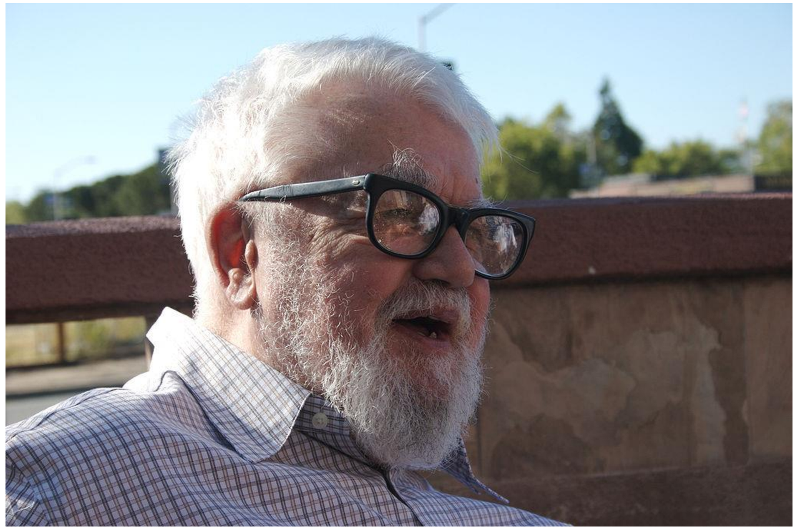
λ

- Invented by Alonzo Church in the 1930’s
- Method to express an anonymous function
- Supported in Java 1.8
- Simplest form: `x -> x*x`
  - Parameter name comes first
  - Then “->” to indicate this is a lambda expression
  - Then an expression to evaluate the result
- Can have multiple parameters: `(x,y)->x*y`
- Can have multiple statements in braces {} with “return”

History of Lambda Expressions

λ

- 1956 Information Processing Langauge
  - Allen Newell, Cliff Shaw, and Herbert Siman at RAND/Carnegie IT
  - List processing (dynamic memory, types, recursion, multi-tasking)
- 1958 LISP
  - John McCarthy at MIT (IBM summer)
  - 2nd major language (after FORTRAN)
  - Mixes data and functions
- 1970 SCHEME
  - LISP dialect using lambdas
  - Guy Steele & Gerald Sussman at MIT



Lambda Concept

- Provide a way to write a function in Java
  - That does not require a class
  - That does not require a method
  - That can be "encapsulated" and passed around like data
  - That is not executed right away… but can be executed when we are ready
- Lambda expressions are a way of writing a function
  - Specify parameters
  - Specify a return value
- Think of a Lambda as a box around code

Lambda Expression Syntax

(*argument_list*) -> *return_expression*;

- *argument_list* : a comma separated list of variable names
  - Types may be unspecified!
  - Types are determined when the lambda expression is used
- *return_expression* : Any java expression
  - Can use variable(s) from the argument list
  - Can also use fields and "final" local variables
  - Expression value is implicitly "returned"

Java "Capture"

- When the lambda expression is created, Java "captures" the value of "this" and keeps it with the lambda expression
- When the lambda expression is evaluated, the CAPTURED value of "this" is used to evaluate the result!
  - This counts as a reference to the object, so garbage collector won't delete the object until (among other things) the lambda expression is unreferenced.
- The actual field value is evaluated at runtime!

Where are Lambda Expressions Used?

- Anywhere an object that implements a “Functional Interfaces” is required
- A "Functional Interface" requires a *single* method
  - For example Runnable, ActionListener, Comparable
  - See java.util.function in the Java library for generic functional interfaces
  - Of course, we can write our own functional interfaces as well

Second Class Lambda Execution!

- In order to execute a lambda expression, we again need to treat the function as a second class citizen
    - Instead of getting the function itself, we get a reference to an "object" that implements a functional interface
    - Use that reference to invoke the method defined by the functional interface
    - The result is that the lambda expression will be evaluated

Functional Programming

- Includes the concept of applying a function "over" a data structure
- E.g. "map"

Haskell

```
map (\x  ->  x* x)  [1, 2, 3, 4]
[1, 4, 9, 16]
```

Scheme

```
> (map (lambda (x) (* x x)) '(1  2 3  4))
(list 1 4 9 16)
```

Python

```
>>> a= [1,2,3,4]
>>> list(map(lambda x: x*x ,a))
[1, 4, 9, 16]
```

C

```
int* map ( int (*f)(int), int len, int array[ ]) {
    int i = 0;      int* ret = (int*)malloc(len*sizeof(int));
    for(i = 0; i < len; i++) { ret[i] = (*f)(array[i]);  }
    return ret; }
```