WIKIPEDIA

Anonymous function

In computer programming, an **anonymous function** (**function literal**, **lambda abstraction**, or **lambda expression**) is a function definition that is not bound to an identifier. Anonymous functions are often arguments being passed to higher-order functions, or used for constructing the result of a higher-order function that needs to return a function.[1] If the function is only used once, or a limited number of times, an anonymous function may be syntactically lighter than using a named function. Anonymous functions are ubiquitous in functional programming languages and other languages with first-class functions, where they fulfil the same role for the function type as literals do for other data types.

Anonymous functions originate in the work of Alonzo Church in his invention of the lambda calculus, in which all functions are anonymous, in 1936, before electronic computers.[2] In several programming languages, anonymous functions are introduced using the keyword *lambda*, and anonymous functions are often referred to as lambdas or lambda abstractions. Anonymous functions have been a feature of programming languages since Lisp in 1958, and a growing number of modern programming languages support anonymous functions.

Contents

Uses

Anonymous functions can be used for containing functionality that need not be named and possibly for short-term use. Some notable examples include closures and currying.

Use of anonymous functions is a matter of style. Using them is never the only way to solve a problem; each anonymous function could instead be defined as a named function and called by name. Some programmers use anonymous functions to encapsulate specific, non-reusable code without littering the code with a lot of little one-line normal functions.

In some programming languages, anonymous functions are commonly implemented for very specific purposes such as binding events to callbacks, or instantiating the function for particular values, which may be more efficient, more readable, and less error-prone than calling a more-generic named function.

The following examples are written in Python 3.

Sorting

When attempting to sort in a non-standard way, it may be easier to contain the sorting logic as an anonymous function instead of creating a named function. Most languages provide a generic sort function that implements a sort algorithm that will sort arbitrary objects. This function usually accepts an arbitrary function that determines how to compare whether two elements equal or if one is greater or less than the other.

Consider sorting a list of strings by length of the string:

```
>>> a = ['house', 'car', 'bike']
>>> a.sort(key=lambda x: len(x))
>>> print(a)
['car', 'bike', 'house']
```

The anonymous function in this example is the lambda expression:

```
lambda x: len(x)
```

The anonymous function accepts one argument, `x`, and returns the length of its argument, which is then used by the `sort()` method as the criteria for sorting. Another example would be sorting items in a list by the name of their class (in Python, everything has a class):

```
>>> a = [10, 'number', 11.2]
>>> a.sort(key=lambda x: x.__class__.__name__)
>>> print(a)
[11.2, 10, 'number']
```

Note that `11.2` has class name "`float`", `10` has class name "`int`", and `'number'` has class name "`str`". The sorted order is "`float`", "`int`", then "`str`".

Closures

Closures are functions evaluated in an environment containing bound variables. The following example binds the variable "threshold" in an anonymous function that compares the input to the threshold.

```
def comp(threshold):
    return lambda x: x < threshold
```

This can be used as a sort of generator of comparison functions:

```
>>> func_a = comp(10)
>>> func_b = comp(20)

>>> print(func_a(5), func_a(8), func_a(13), func_a(21))
True True False False

>>> print(func_b(5), func_b(8), func_b(13), func_b(21))
True True True False
```

It would be impractical to create a function for every possible comparison function and may be too inconvenient to keep the threshold around for further use. Regardless of the reason why a closure is used, the anonymous function is the entity that contains the functionality that does the comparing.

Currying

Currying is the process of changing a function so that it takes fewer inputs (in this case, transforming a function that performs division by any integer into one that performs division by a set integer).

```
>>> def divide(x, y):
...     return x / y

>>> def divisor(d):
...     return lambda x: divide(x, d)

>>> half = divisor(2)
>>> third = divisor(3)

>>> print(half(32), third(32))
16.0 10.666666666666666

>>> print(half(40), third(40))
20.0 13.333333333333334
```

While the use of anonymous functions is perhaps not common with currying, it still can be used. In the above example, the function divisor generates functions with a specified divisor. The functions half and third curry the divide function with a fixed divisor.

The divisor function also forms a closure by binding the "d" variable.

Higher-order functions

Python 3 includes several functions that take anonymous functions as an argument. This section describes some of them.

Map

The map function performs a function call on each element of a list. The following example squares every element in an array with an anonymous function.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> print(list(map(lambda x: x*x, a)))
[1, 4, 9, 16, 25, 36]
```

The anonymous function accepts an argument and multiplies it by itself (squares it). Note that the above form is discouraged by the creators of the language, who maintain that the form presented below has the same meaning and is more aligned with the philosophy of the language:

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> print([x*x for x in a])
[1, 4, 9, 16, 25, 36]
```

Filter

The filter function returns all elements from a list that evaluate True when passed to a certain function.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> print(list(filter(lambda x: x % 2 == 0, a)))
[2, 4, 6]
```

The anonymous function checks if the argument passed to it is even. The same as with map form below is considered as more appropriate:

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> print([x for x in a if x % 2 == 0])
[2, 4, 6]
```

Fold

The fold/reduce function runs over all elements in a list (usually left-to-right), accumulating a value as it goes. A common use of this is to combine all elements of a list into one value, for example:

```
>>> a = [1, 2, 3, 4, 5]
>>> print(list(reduce(lambda x,y: x*y, a)))
120
```

This performs

$$(((1 \times 2) \times 3) \times 4) \times 5 = 120.$$

The anonymous function here is the multiplication of the two arguments.

The result of a fold need not be one value. Instead, both map and filter can be created using fold. In map, the value that is accumulated is a new list, containing the results of applying a function to each element of the original list. In filter, the value that is accumulated is a new list containing only those elements that match the given condition.

List of languages

The following is a list of programming languages that support unnamed anonymous functions fully, or partly as some variant, or not at all.

This table shows some general trends. First, the languages that do not support anonymous functions (C, Pascal, Object Pascal) are all statically typed languages. However, statically typed languages can support anonymous functions. For example, the ML languages are statically typed and fundamentally include anonymous functions, and Delphi, a dialect of Object Pascal, has been extended to support anonymous functions, as has C++ (by the C++11 standard). Second, the languages that treat functions as first-class functions (Dylan, Haskell, JavaScript, Lisp, ML, Perl, Python, Ruby, Scheme) generally have anonymous function support so that functions can be defined and passed around as easily as other data types.

Language	Support	Notes
ActionScript	✓	
Ada	✗	Expression functions are a part of Ada2012
ALGOL 68	✓	
APL	✓	Dyalog, ngn and dzaima APL fully support both dfns and tacit functions. GNU APL has rather limited support for dfns.
Assembly Language	✗	
Bash	✓	A library has been made to support anonymous functions in Bash.[3]
C	✗	Support is provided in Clang and along with the LLVM compiler-rt lib. GCC support is given for a macro implementation which enables the possibility of use. See below for more details.
C#	✓	
C++	✓	As of the C++11 standard
CFML	✓	As of Railo 4,[4] ColdFusion 10[5]
Clojure	✓	
COBOL	✗	Micro Focus's non-standard Managed COBOL dialect supports lambdas, which are called anonymous delegates/methods.[6]
Curl	✓	
D	✓	
Dart	✓	
Delphi	✓	
Dylan	✓	
Eiffel	✓	
Elm	✓	
Elixir	✓	
Erlang	✓	
F#	✓	
Factor	✓	"Quotations" support this[7]
Fortran	✗	
Frink	✓	
Go	✓	
Gosu	✓	[8]
Groovy	✓	[9]
Haskell	✓	
Haxe	✓	
Java	✓	Supported in Java 8. See the Java Limitations section below for details.

JavaScript	✓	
Julia	✓	
Kotlin	✓	
Lisp	✓	
Logtalk	✓	
Lua	✓	
MUMPS	✗	
Mathematica	✓	
Maple	✓	
MATLAB	✓	
Maxima	✓	
OCaml	✓	
Octave	✓	
Object Pascal	✓	Delphi, a dialect of Object Pascal, supports anonymous functions (formally, *anonymous methods*) natively since Delphi 2009. The Oxygene Object Pascal dialect also supports them.
Objective-C (Mac OS X 10.6+)	✓	called blocks; in addition to Objective-C, blocks can also be used on C and C++ when programming on Apple's platform
Pascal	✗	
Perl	✓	
PHP	✓	As of PHP 5.3.0, true anonymous functions are supported. Formerly, only partial anonymous functions were supported, which worked much like C#'s implementation.
PL/I	✗	
Python	✓	Python supports anonymous functions through the lambda syntax, which supports only expressions, not statements.
R	✓	
Racket	✓	
Rexx	✗	
RPG	✗	
Ruby	✓	Ruby's anonymous functions, inherited from Smalltalk, are called blocks.
Rust	✓	
Scala	✓	
Scheme	✓	
Smalltalk	✓	Smalltalk's anonymous functions are called blocks.
Standard ML	✓	
Swift	✓	Swift's anonymous functions are called Closures.
TypeScript	✓	

Tcl	✓	
Vala	✓	
Visual Basic .NET v9	✓	
Visual Prolog v 7.2	✓	
Wolfram Language	✓	

Examples

Numerous languages support anonymous functions, or something similar.

C (non-standard extension)

The anonymous function is not supported by standard C programming language, but supported by some C dialects, such as *GCC* and *Clang*.

GCC

GNU Compiler Collection (GCC) supports anonymous functions, mixed by nested functions and statement expressions. It has the form:

```
( { return_type anonymous_functions_name (parameters) { function_body } anonymous_functions_name; } )
```

The following example works only with GCC. Because of how macros are expanded, the `l_body` cannot contain any commas outside of parentheses; GCC treats the comma as a delimiter between macro arguments. The argument `l_ret_type` can be removed if `__typeof__` is available; in the example below using `__typeof__` on array would return testtype *, which can be dereferenced for the actual value if needed.

```
#include <stdio.h>

//* this is the definition of the anonymous function */
#define lambda(l_ret_type, l_arguments, l_body)        \
  ({                                                   \
    l_ret_type l_anonymous_functions_name l_arguments  \
    l_body                                             \
    &l_anonymous_functions_name;                       \
    })

#define forEachInArray(fe_arrType, fe_arr, fe_fn_body)                                  \
{                                                                                       \
  int i=0;                                                                              \
  for(;i<sizeof(fe_arr)/sizeof(fe_arrType);i++) {  fe_arr[i] = fe_fn_body(&fe_arr[i]); }  \
}

typedef struct __test
{
  int a;
  int b;
} testtype;

void printout(const testtype * array)
{
  int i;
  for ( i = 0; i < 3; ++ i )
```

```
    printf("%d %d\n", array[i].a, array[i].b);
  printf("\n");
}

int main(void)
{
  testtype array[] = { {0,1}, {2,3}, {4,5} };

  printout(array);
  /* the anonymous function is given as function for the foreach */
  forEachInArray(testtype, array,
    lambda (testtype, (void *item),
    {
      int temp = (*( testtype *) item).a;
      (*( testtype *) item).a = (*( testtype *) item).b;
      (*( testtype *) item).b = temp;
      return (*( testtype *) item);
    }));
  printout(array);
  return 0;
}
```

Clang (C, C++, Objective-C, Objective-C++)

Clang supports anonymous functions, called blocks, which have the form:

```
^return_type ( parameters ) { function_body }
```

The type of the blocks above is `return_type (^)(parameters)`.

Using the aforementioned *blocks* extension and Grand Central Dispatch (libdispatch), the code could look simpler:

```
#include <stdio.h>
#include <dispatch/dispatch.h>

int main(void) {
  void (^count_loop)() = ^{
    for (int i = 0; i < 100; i++)
      printf("%d\n", i);
    printf("ah ah ah\n");
  };

/* Pass as a parameter to another function */
  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), count_loop);

/* Invoke directly */
  count_loop();

  return 0;
}
```

The code with blocks should be compiled with `-fblocks` and linked with `-lBlocksRuntime`

C++ (since C++11)

C++11 supports anonymous functions, called *lambda expressions*, which have the form:

```
[capture](parameters) -> return_type { function_body }
```

An example lambda function is defined as follows:

```
[](int x, int y) -> int { return x + y; }
```

C++11 also supports closures. Closures are defined between square brackets [and] in the declaration of lambda expression. The mechanism allows these variables to be captured by value or by reference. The following table demonstrates this:

```
[]        //no variables defined. Attempting to use any external variables in the lambda is an error.
[x, &y]   //x is captured by value, y is captured by reference
[&]       //any external variable is implicitly captured by reference if used
[=]       //any external variable is implicitly captured by value if used
[&, x]    //x is explicitly captured by value. Other variables will be captured by reference
[=, &z]   //z is explicitly captured by reference. Other variables will be captured by value
```

Variables captured by value are constant by default. Adding `mutable` after the parameter list makes them non-constant.

The following two examples demonstrate use of a lambda expression:

```
std::vector<int> some_list{ 1, 2, 3, 4, 5 };
int total = 0;
std::for_each(begin(some_list), end(some_list), [&total](int x) {
    total += x;
});
```

This computes the total of all elements in the list. The variable `total` is stored as a part of the lambda function's closure. Since it is a reference to the stack variable `total`, it can change its value.

```
std::vector<int> some_list{ 1, 2, 3, 4, 5 };
int total = 0;
int value = 5;
std::for_each(begin(some_list), end(some_list), [&, value, this](int x) {
    total += x * value * this->some_func();
});
```

This will cause `total` to be stored as a reference, but `value` will be stored as a copy.

The capture of `this` is special. It can only be captured by value, not by reference. `this` can only be captured if the closest enclosing function is a non-static member function. The lambda will have the same access as the member that created it, in terms of protected/private members.

If `this` is captured, either explicitly or implicitly, then the scope of the enclosed class members is also tested. Accessing members of `this` does not need explicit use of `this->` syntax.

The specific internal implementation can vary, but the expectation is that a lambda function that captures everything by reference will store the actual stack pointer of the function it is created in, rather than individual references to stack variables. However, because most lambda functions are small and local in scope, they are likely candidates for inlining, and thus need no added storage for references.

If a closure object containing references to local variables is invoked after the innermost block scope of its creation, the behaviour is undefined.

Lambda functions are function objects of an implementation-dependent type; this type's name is only available to the compiler. If the user wishes to take a lambda function as a parameter, the parameter type must be a template type, or they must create a `std::function` or a similar object to capture the lambda value. The use of the `auto` keyword can help store the lambda function,

```
auto my_lambda_func = [&](int x) { /*...*/ };
auto my_onheap_lambda_func = new auto([=](int x) { /*...*/ });
```

Here is an example of storing anonymous functions in variables, vectors, and arrays; and passing them as named parameters:

```
#include <functional>
#include <vector>
#include <iostream>

double eval(std::function <double(double)> f, double x = 2.0)
{
    return f(x);
}

int main()
{
    std::function<double(double)> f0    = [](double x){return 1;};
    auto                          f1    = [](double x){return x;};
    decltype(f0)                  fa[3] = {f0,f1,[](double x){return x*x;}};
    std::vector<decltype(f0)>     fv    = {f0,f1};
    fv.push_back                  ([](double x){return x*x;});
    for(int i=0;i<fv.size();i++)
        std::cout << fv[i](2.0) << std::endl;
    for(int i=0;i<3;i++)
        std::cout << fa[i](2.0) << std::endl;
    for(auto &f : fv)
        std::cout << f(2.0) << std::endl;
    for(auto &f : fa)
        std::cout << f(2.0) << std::endl;
    std::cout << eval(f0) << std::endl;
    std::cout << eval(f1) << std::endl;
    std::cout << eval([](double x){return x*x;}) << std::endl;
    return 0;
}
```

A lambda expression with an empty capture specification ([]) can be implicitly converted into a function pointer with the same type as the lambda was declared with. So this is legal:

```
auto a_lambda_func = [](int x) { /*...*/ };
void (* func_ptr)(int) = a_lambda_func;
func_ptr(4); //calls the lambda.
```

The Boost library provides its own syntax for lambda functions as well, using the following syntax:[10]

```
for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

C#

In C#, support for anonymous functions has deepened through the various versions of the language compiler. The language v3.0, released in November 2007 with .NET Framework v3.5, has full support of anonymous functions. C# names them *lambda expressions*, following the original version of anonymous functions, the lambda calculus.[11]

```
// the first int is the x' type
// the second int is the return type
// <see href="http://msdn.microsoft.com/en-us/library/bb549151.aspx" />
Func<int,int> foo = x => x*x;
Console.WriteLine(foo(7));
```

While the function is anonymous, it cannot be assigned to an implicitly typed variable, because the lambda syntax may be used for denoting an anonymous function or an expression tree, and the choice cannot automatically be decided by the compiler. E.g., this does not work:

```
// will NOT compile!
var foo = (int x) => x*x;
```

However, a lambda expression can take part in type inference and can be used as a method argument, e.g. to use anonymous functions with the Map capability available with `System.Collections.Generic.List` (in the `ConvertAll()` method):

```
// Initialize the list:
var values = new List<int>() { 7, 13, 4, 9, 3 };
// Map the anonymous function over all elements in the list, return the new list
var foo = values.ConvertAll(d => d*d) ;
// the result of the foo variable is of type System.Collections.Generic.List<Int32>
```

Prior versions of C# had more limited support for anonymous functions. C# v1.0, introduced in February 2002 with the .NET Framework v1.0, provided partial anonymous function support through the use of delegates. This construct is somewhat similar to PHP delegates. In C# 1.0, delegates are like function pointers that refer to an explicitly named method within a class. (But unlike PHP, the name is unneeded at the time the delegate is used.) C# v2.0, released in November 2005 with the .NET Framework v2.0, introduced the concept of anonymous methods as a way to write unnamed inline statement blocks that can be executed in a delegate invocation. C# 3.0 continues to support these constructs, but also supports the lambda expression construct.

This example will compile in C# 3.0, and exhibits the three forms:

```
   public class TestDriver
   {
       delegate int SquareDelegate(int d);
       static int Square(int d)
       {
           return d*d;
       }

       static void Main(string[] args)
       {
           // C# 1.0: Original delegate syntax needed
           // initializing with a named method.
           SquareDelegate A = new SquareDelegate(Square);
           System.Console.WriteLine(A(3));

           // C# 2.0: A delegate can be initialized with
           // inline code, called an "anonymous method". This
           // method takes an int as an input parameter.
           SquareDelegate B = delegate(int d) { return d*d; };
           System.Console.WriteLine(B(5));

           // C# 3.0. A delegate can be initialized with
           // a lambda expression. The lambda takes an int, and returns an int.
           // The type of x is inferred by the compiler.
           SquareDelegate C = x => x*x;
           System.Console.WriteLine(C(7));

           // C# 3.0. A delegate that accepts one input and
           // returns one output can also be implicitly declared with the Func<> type.
           System.Func<int,int> D = x => x*x;
           System.Console.WriteLine(D(9));
       }
   }
```

In the case of the C# 2.0 version, the C# compiler takes the code block of the anonymous function and creates a static private function. Internally, the function gets a generated name, of course; this generated name is based on the name of the method in which the Delegate is declared. But the name is not exposed to application code except by using reflection.

In the case of the C# 3.0 version, the same mechanism applies.

ColdFusion Markup Language (CFML)

```
fn = function(){
  // statements
};
```

CFML supports any statements within the function's definition, not simply expressions.

CFML supports recursive anonymous functions:

```
factorial = function(n){
    return n > 1 ? n * factorial(n-1) : 1;
};
```

CFML anonymous functions implement closure.

D

D uses inline delegates to implement anonymous functions. The full syntax for an inline delegate is

```
return_type delegate(arguments){/*body*/}
```

If unambiguous, the return type and the keyword *delegate* can be omitted.

```
(x){return x*x;}
delegate (x){return x*x;} // if more verbosity is needed
(int x){return x*x;} // if parameter type cannot be inferred
delegate (int x){return x*x;} // ditto
delegate double(int x){return x*x;} // if return type must be forced manually
```

Since version 2.0, D allocates closures on the heap unless the compiler can prove it is unnecessary; the `scope` keyword can be used for forcing stack allocation. Since version 2.058, it is possible to use shorthand notation:

```
x => x*x;
(int x) => x*x;
(x,y) => x*y;
(int x, int y) => x*y;
```

An anonymous function can be assigned to a variable and used like this:

```
auto sqr = (double x){return x*x;};
double y = sqr(4);
```

Dart

Dart supports anonymous functions.

```
var sqr = (x) => x * x;
print(sqr(5));
```

or

```
print(((x) => x * x)(5));
```

Delphi

Delphi introduced anonymous functions in version 2009.

```
program demo;

type
  TSimpleProcedure = reference to procedure;
  TSimpleFunction = reference to function(x: string): Integer;

var
  x1: TSimpleProcedure;
  y1: TSimpleFunction;

begin
  x1 := procedure
    begin
      Writeln('Hello World');
    end;
  x1;   //invoke anonymous method just defined

  y1 := function(x: string): Integer
    begin
      Result := Length(x);
    end;
  Writeln(y1('bar'));
end.
```

PascalABC.NET

PascalABC.NET supports anonymous functions using lambda syntax

```
begin
  var n := 10000000;
  var pp := Range(1,n)
    .Select(x->Rec(Random(),Random()))
    .Where(p->sqr(p.Item1)+sqr(p.Item2)<1)
    .Count/n*4;
  Print(pp);
end.
```

Elixir

Elixir uses the closure `fn` for anonymous functions.

```
sum = fn(a, b) -> a + b end
sum.(4, 3)
#=> 7
```

```
square = fn(x) -> x * x end
Enum.map [1, 2, 3, 4], square
#=> [1, 4, 9, 16]
```

Erlang

Erlang uses a syntax for anonymous functions similar to that of named functions.

```
% Anonymous function bound to the Square variable
Square = fun(X) -> X * X end.

% Named function with the same functionality
square(X) -> X * X.
```

Go

Go supports anonymous functions.

```
foo := func(x int) int {
    return x * x
}
fmt.Println(foo(10))
```

Haskell

Haskell uses a concise syntax for anonymous functions (lambda expressions).

```
\x -> x * x
```

Lambda expressions are fully integrated with the type inference engine, and support all the syntax and features of "ordinary" functions (except for the use of multiple definitions for pattern-matching, since the argument list is only specified once).

```
map (\x -> x * x) [1..5] -- returns [1, 4, 9, 16, 25]
```

The following are all equivalent:

```
f x y = x + y
f x = \y -> x + y
f = \x y -> x + y
```

Haxe

In Haxe, anonymous functions are called lambda, and use the syntax `function(argument-list) expression;` .

```
var f = function(x) return x*x;
f(8); // 64

(function(x,y) return x+y)(5,6); // 11
```

Java

Java supports anonymous functions, named *Lambda Expressions*, starting with JDK 8.[12]

A lambda expression consists of a comma separated list of the formal parameters enclosed in parentheses, an arrow token (->), and a body. Data types of the parameters can always be omitted, as can the parentheses if there is only one parameter. The body can consist of one statement or a statement block.[13]

```
// with no parameter
() -> System.out.println("Hello, world.")

// with one parameter (this example is an identity function).
a -> a

// with one expression
(a, b) -> a + b

// with explicit type information
(long id, String name) -> "id: " + id + ", name:" + name

// with a code block
(a, b) -> { return a + b; }

// with multiple statements in the lambda body. It needs a code block.
// This example also includes two nested lambda expressions (the first one is also a closure).
(id, defaultPrice) -> {
  Optional<Product> product = productList.stream().filter(p -> p.getId() == id).findFirst();
  return product.map(p -> p.getPrice()).orElse(defaultPrice);
}
```

Lambda expressions are converted to "functional interfaces" (defined as interfaces that contain only one abstract method in addition to one or more default or static methods),[13] as in the following example:

```
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);

        default IntegerMath swap() {
          return (a, b) -> operation(b, a);
        }
    }

    private static int apply(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " + apply(40, 2, addition));
        System.out.println("20 - 10 = " + apply(20, 10, subtraction));
        System.out.println("10 - 20 = " + apply(20, 10, subtraction.swap()));
    }
}
```

In this example, a functional interface called `IntegerMath` is declared. Lambda expressions that implement `IntegerMath` are passed to the `apply()` method to be executed. Default methods like `swap` define methods on functions.

Java 8 introduced another mechanism named method reference (the :: operator) to create a lambda on an existing method. A method reference doesn't indicate the number or types of arguments because those are extracted from the abstract method of the functional interface.

```
IntBinaryOperator sum = Integer::sum;
```

In the example above, the functional interface `IntBinaryOperator` declares an abstract method `int applyAsInt(int, int)`, so the compiler looks for a method `int sum(int, int)` in the class `java.lang.Integer`.

Java Limitations

Java 8 lambdas have the following limitations:

- Lambdas can throw checked exceptions, but such lambdas will not work with the interfaces used by the Collection API.
- Variables that are in-scope where the lambda is declared may only be accessed inside the lambda if they are effectively final, i.e. if the variable is not mutated inside or outside of the lambda scope.

JavaScript

JavaScript/ECMAScript supports anonymous functions.

```
alert((function(x){
    return x*x;
})(10));
```

ES6 supports "arrow function" syntax, where a `=>` symbol separates the anonymous function's parameter list from the body:

```
alert((x => x*x)(10));
```

This construct is often used in Bookmarklets. For example, to change the title of the current document (visible in its window's title bar) to its URL, the following bookmarklet may seem to work.

```
javascript:document.title=location.href;
```

However, as the assignment statement returns a value (the URL itself), many browsers actually create a new page to display this value.

Instead, an anonymous function, that does not return a value, can be used:

```
javascript:(function(){document.title=location.href;})();
```

The function statement in the first (outer) pair of parentheses declares an anonymous function, which is then executed when used with the last pair of parentheses. This is almost equivalent to the following, which populates the environment with `f` unlike an anonymous function.

```
javascript:var f = function(){document.title=location.href;}; f();
```

Use void() to avoid new pages for arbitrary anonymous functions:

```
javascript:void(function(){return document.title=location.href;}());
```

or just:

```
javascript:void(document.title=location.href);
```

JavaScript has syntactic subtleties for the semantics of defining, invoking and evaluating anonymous functions. These subliminal nuances are a direct consequence of the evaluation of parenthetical expressions. The following constructs which are called immediately-invoked function expression illustrate this:

```
(function(){ ... }())
```

and

```
(function(){ ... })()
```

Representing "`function(){ ... }`" by `f`, the form of the constructs are a parenthetical within a parenthetical `(f())` and a parenthetical applied to a parenthetical `(f)()`.

Note the general syntactic ambiguity of a parenthetical expression, parenthesized arguments to a function and the parentheses around the formal parameters in a function definition. In particular, JavaScript defines a `,` (comma) operator in the context of a parenthetical expression. It is no mere coincidence that the syntactic forms coincide for an expression and a function's arguments (ignoring the function formal parameter syntax)! If `f` is not identified in the constructs above, they become `(())` and `()()`. The first provides no syntactic hint of any resident function but the second MUST evaluate the first parenthetical as a function to be legal JavaScript. (Aside: for instance, the `()`'s could be ([],{},42,"abc",function() {}) as long as the expression evaluates to a function.)

Also, a function is an Object instance (likewise objects are Function instances) and the object literal notation brackets, `{}` for braced code, are used when defining a function this way (as opposed to using `new Function(...)`). In a very broad non-rigorous sense (especially since global bindings are compromised), an arbitrary sequence of braced JavaScript statements, `{stuff}`, can be considered to be a fixed point of

```
(function(){{ function(){{ ... {{ function(){stuff}() )} ... )}() )}() )
```

More correctly but with caveats,

```
( function(){stuff}() ) ~=
   A_Fixed_Point_of(
      function(){ return function(){ return ... { return function(){stuff}() } ... }() }()
   )
```

Note the implications of the anonymous function in the JavaScript fragments that follow:

- `function(){ ... }()` without surrounding `()`'s is generally not legal
- `(f=function(){ ... })` does not "forget" `f` globally unlike `(function f(){ ... })`

> Performance metrics to analyze the space and time complexities of function calls, call stack, etc. in a JavaScript interpreter engine implement easily with these last anonymous function constructs. From the implications of the results, it is possible to deduce some of an engine's recursive versus iterative implementation details, especially tail-recursion.

Julia

In Julia anonymous functions are defined using the syntax `(arguments)->(expression)`,

```
julia> f = x -> x*x; f(8)
64
julia> ((x,y)->x+y)(5,6)
11
```

Lisp

Lisp and Scheme support anonymous functions using the "lambda" construct, which is a reference to lambda calculus. Clojure supports anonymous functions with the "fn" special form and #() reader syntax.

```
(lambda (arg) (* arg arg))
```

Common Lisp

Common Lisp has the concept of lambda expressions. A lambda expression is written as a list with the symbol "lambda" as its first element. The list then contains the argument list, documentation or declarations and a function body. Lambda expressions can be used inside lambda forms and with the special operator "function".

```
(function (lambda (arg) (do-something arg)))
```

"function" can be abbreviated as #'. Also, macro *lambda* exists, which expands into a function form:

```
; using sharp quote
#'(lambda (arg) (do-something arg))
; using the lambda macro:
(lambda (arg) (do-something arg))
```

One typical use of anonymous functions in Common Lisp is to pass them to higher-order functions like *mapcar*, which applies a function to each element of a list and returns a list of the results.

```
(mapcar #'(lambda (x) (* x x))
        '(1 2 3 4))
; -> (1 4 9 16)
```

The *lambda form* in Common Lisp allows a *lambda expression* to be written in a function call:

```
((lambda (x y)
   (+ (sqrt x) (sqrt y)))
 10.0
 12.0)
```

Anonymous functions in Common Lisp can also later be given global names:

```
(setf (symbol-function 'sqr)
      (lambda (x) (* x x)))
; which allows us to call it using the name SQR:
(sqr 10.0)
```

Scheme

Scheme's *named functions* is simply syntactic sugar for anonymous functions bound to names:

```
(define (somename arg)
  (do-something arg))
```

expands (and is equivalent) to

```
(define somename
  (lambda (arg)
    (do-something arg)))
```

Clojure

Clojure supports anonymous functions through the "fn" special form:

```
(fn [x] (+ x 3))
```

There is also a reader syntax to define a lambda:

```
# (+ % %2%3) ; Defines an anonymous function that takes three arguments and sums them.
```

Like Scheme, Clojure's "named functions" are simply syntactic sugar for lambdas bound to names:

```
(defn func [arg] (+ 3 arg))
```

expands to:

```
(def func (fn [arg] (+ 3 arg)))
```

Lua

In Lua (much as in Scheme) all functions are anonymous. A *named function* in Lua is simply a variable holding a reference to a function object.[14]

Thus, in Lua

```
function foo(x) return 2*x end
```

is just syntactical sugar for

```
foo = function(x) return 2*x end
```

An example of using anonymous functions for reverse-order sorting:

```
table.sort(network, function(a,b)
  return a.name > b.name
end)
```

Wolfram Language, Mathematica

The Wolfram Language is the programming language of Mathematica. Anonymous functions are important in programming the latter. There are several ways to create them. Below are a few anonymous functions that increment a number. The first is the most common. `#1` refers to the first argument and `&` marks the end of the anonymous function.

```
#1+1&
Function[x,x+1]
x \[Function] x+1
```

So, for instance:

```
f:= #1^2&;f[8]
 64
#1+#2&[5,6]
 11
```

Also, Mathematica has an added construct to make recursive anonymous functions. The symbol '`#0`' refers to the entire function. The following function calculates the factorial of its input:

```
If[#1 == 1, 1, #1 * #0[#1-1]]&
```

For example, 6 factorial would be:

```
If[#1 == 1, 1, #1 * #0[#1-1]]&[6]
720
```

MATLAB, Octave

Anonymous functions in MATLAB or Octave are defined using the syntax `@(argument-list)expression`. Any variables that are not found in the argument list are inherited from the enclosing scope.

```
> f = @(x)x*x; f(8)
ans =  64
> (@(x,y)x+y)(5,6) % Only works in Octave
ans =  11
```

Maxima

In Maxima anonymous functions are defined using the syntax `lambda(argument-list,expression)`,

```
f: lambda([x],x*x); f(8);
64

lambda([x,y],x+y)(5,6);
11
```

ML

The various dialects of ML support anonymous functions.

OCaml

```
fun arg -> arg * arg
```

F#

F# supports anonymous functions, as follows:

```
(fun x -> x * x) 20 // 400
```

Standard ML

Standard ML supports anonymous functions, as follows:

```
fn  arg => arg * arg
```

Perl

Perl 5

Perl 5 supports anonymous functions, as follows:

```
(sub { print "I got called\n" })->();         # 1. fully anonymous, called as created

my $squarer = sub { my $x = shift; $x * $x }; # 2. assigned to a variable

sub curry {
    my ($sub, @args) = @_;
    return sub { $sub->(@args, @_) };         # 3. as a return value of another function
}

# example of currying in Perl programming
sub sum { my $tot = 0; $tot += $_ for @_; $tot } # returns the sum of its arguments
my $curried = curry \&sum, 5, 7, 9;
print $curried->(1,2,3), "\n";    # prints 27 ( = 5 + 7 + 9 + 1 + 2 + 3 )
```

Other constructs take *bare blocks* as arguments, which serve a function similar to lambda functions of one parameter, but don't have the same parameter-passing convention as functions -- @_ is not set.

```
my @squares = map { $_ * $_ } 1..10;   # map and grep don't use the 'sub' keyword
my @square2 = map $_ * $_, 1..10;      # braces unneeded for one expression

my @bad_example = map { print for @_ } 1..10; # values not passed like normal Perl function
```

Perl 6

In Perl 6, all blocks (even those associated with if, while, etc.) are anonymous functions. A block that is not used as an rvalue is executed immediately.

1. fully anonymous, called as created

```
{ say "I got called" };
```

2. assigned to a variable

```
my $squarer1 = -> $x { $x * $x };             # 2a. pointy block
my $squarer2 = { $^x * $^x };                 # 2b. twigil
my $squarer3 = { my $x = shift @_; $x * $x }; # 2c. Perl 5 style
```

3. currying

```
sub add ($m, $n) { $m + $n }
my $seven   = add(3, 4);
my $add_one = &add.assuming(m => 1);
my $eight   = $add_one($seven);
```

4. WhateverCode object

```
my $w = * - 1;       # WhateverCode object
my $b = { $_ - 1 };  # same functionality, but as Callable block
```

PHP

Before 4.0.1, PHP had no anonymous function support.[15]

PHP 4.0.1 to 5.3

PHP 4.0.1 introduced the `create_function` which was the initial anonymous function support. This function call makes a new randomly named function and returns its name (as a string)

```
$foo = create_function('$x', 'return $x*$x;');
$bar = create_function("\$x", "return \$x*\$x;");
echo $foo(10);
```

The argument list and function body must be in single quotes, or the dollar signs must be escaped. Otherwise, PHP assumes "$x" means the variable $x and will substitute it into the string (despite possibly not existing) instead of leaving "$x" in the string. For functions with quotes or functions with lots of variables, it can get quite tedious to ensure the intended function body is what PHP interprets.

Each invocation of `create_function` makes a new function, which exists for the rest of the program, and cannot be *garbage collected*, using memory in the program irreversibly. If this is used to create anonymous functions many times, e.g., in a loop, it can cause problems such as memory bloat.

PHP 5.3

PHP 5.3 added a new class called `Closure` and magic method `__invoke()` that makes a class instance invocable.[16]

```
$x = 3;
$func = function($z) { return $z *= 2; };
echo $func($x); // prints 6
```

In this example, `$func` is an instance of `Closure` and `echo $func($x)` is equivalent to `echo $func->__invoke($x)`. PHP 5.3 mimics anonymous functions but it does not support true anonymous functions because PHP functions are still not first-class objects.

PHP 5.3 does support closures but the variables must be explicitly indicated as such:

```
$x = 3;
$func = function() use(&$x) { $x *= 2; };
$func();
echo $x; // prints 6
```

The variable `$x` is bound by reference so the invocation of `$func` modifies it and the changes are visible outside of the function.

Prolog's dialects

Logtalk

Logtalk uses the following syntax for anonymous predicates (lambda expressions):

```
{FreeVar1, FreeVar2, ...}/[LambdaParameter1, LambdaParameter2, ...]>>Goal
```

A simple example with no free variables and using a list mapping predicate is:

```
| ?- meta::map([X,Y]>>(Y is 2*X), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

Currying is also supported. The above example can be written as:

```
| ?- meta::map([X]>>([Y]>>(Y is 2*X)), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

Visual Prolog

Anonymous functions (in general anonymous *predicates*) were introduced in Visual Prolog in version 7.2.[17] Anonymous predicates can capture values from the context. If created in an object member, it can also access the object state (by capturing `This`).

`mkAdder` returns an anonymous function, which has captured the argument `X` in the closure. The returned function is a function that adds `X` to its argument:

```
clauses
    mkAdder(X) = { (Y) = X+Y }.
```

Python

Python supports simple anonymous functions through the lambda form. The executable body of the lambda must be an expression and can't be a statement, which is a restriction that limits its utility. The value returned by the lambda is the value of the contained expression. Lambda forms can be used anywhere ordinary functions can. However these restrictions make it a very limited version of a normal function. Here is an example:

```
>>> foo = lambda x: x*x
>>> print(foo(10))
100
```

In general, Python convention encourages the use of named functions defined in the same scope as one might typically use an anonymous functions in other languages. This is acceptable as locally defined functions implement the full power of closures and are almost as efficient as the use of a lambda in Python. In this example, the built-in power function can be said to have been curried:

```
>>> def make_pow(n):
...     def fixed_exponent_pow(x):
...         return pow(x, n)
...     return fixed_exponent_pow
...
>>> sqr = make_pow(2)
>>> print (sqr(10))
100
>>> cub = make_pow(3)
>>> print (cub(10))
1000
```

R

In R the anonymous functions are defined using the syntax `function(argument-list)expression` .

```
> f <- function(x)x*x; f(8)
[1] 64
> (function(x,y)x+y)(5,6)
[1] 11
```

Ruby

Ruby supports anonymous functions by using a syntactical structure called *block*. There are two data types for blocks in Ruby. `Procs` behave similarly to closures, whereas `lambdas` behave more analogous to an anonymous function.[18] When passed to a method, a block is converted into a Proc in some circumstances.

```
irb(main):001:0> # Example 1:
irb(main):002:0* # Purely anonymous functions using blocks.
irb(main):003:0* ex = [16.2, 24.1, 48.3, 32.4, 8.5]
=> [16.2, 24.1, 48.3, 32.4, 8.5]
irb(main):004:0> ex.sort_by { |x| x - x.to_i } # Sort by fractional part, ignoring integer part.
=> [24.1, 16.2, 48.3, 32.4, 8.5]
irb(main):005:0> # Example 2:
irb(main):006:0* # First-class functions as an explicit object of Proc -
irb(main):007:0* ex = Proc.new { puts "Hello, world!" }
=> #<Proc:0x007ff4598705a0@(irb):7>
irb(main):008:0> ex.call
Hello, world!
=> nil
irb(main):009:0> # Example 3:
irb(main):010:0* # Function that returns lambda function object with parameters
irb(main):011:0* def is_multiple_of(n)
irb(main):012:1>   lambda{|x| x % n == 0}
```

```
irb(main):013:1> end
=> nil
irb(main):014:0> multiple_four = is_multiple_of(4)
=> #<Proc:0x007ff458b45f88@(irb):12 (lambda)>
irb(main):015:0> multiple_four.call(16)
=> true
irb(main):016:0> multiple_four[15]
=> false
```

Scala

In Scala, anonymous functions use the following syntax:[19]

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

In certain contexts, like when an anonymous function is a parameter being passed to another function, the compiler can infer the types of the parameters of the anonymous function and they can be omitted in the syntax. In such contexts, it is also possible to use a shorthand for anonymous functions using the underscore character to introduce unnamed parameters.

```
val list = List(1, 2, 3, 4)
list.reduceLeft( (x, y) => x + y )
// Here, the compiler can infer that the types of x and y are both Int.
// Thus, it needs no type annotations on the parameters of the anonymous function.

list.reduceLeft( _ + _ )
// Each underscore stands for a new unnamed parameter in the anonymous function.
// This results in an even shorter equivalent to the anonymous function above.
```

Smalltalk

In Smalltalk anonymous functions are called blocks and they are invoked (called) by sending them a "value" message. If arguments are to be passed, a "value:...value:" message with a corresponding number of value arguments must be used.

```
[:x | x*x ] value: 4
"returns 16"
```

Smalltalk blocks are technically closures, allowing them to outlive their defining scope and still refer to the variables declared therein.

```
[:a |
    [:n | a + n ]
] value: 10
"returns the inner block, which adds 10 to its argument."
```

Swift

In Swift, anonymous functions are called closures.[20] The syntax has following form:

```
{ (parameters) -> returnType in
  statement
}
```

For example:

```
{ (s1: String, s2: String) -> Bool in
  return s1 > s2
}
```

For sake of brevity and expressiveness, the parameter types and return type can be omitted if these can be inferred:

```
{ s1, s2 in return s1 > s2 }
```

Similarly, Swift also supports implicit return statements for one-statement closures:

```
{ s1, s2 in s1 > s2 }
```

Finally, the parameter names can be omitted as well; when omitted, the parameters are referenced using shorthand argument names, consisting of the $ symbol followed by their position (e.g. $0, $1, $2, etc.):

```
{ $0 > $1 }
```

Tcl

In Tcl, applying the anonymous squaring function to 2 looks as follows:[21]

```
apply {x {expr {$x*$x}}} 2
# returns 4
```

This example involves two candidates for what it means to be a *function* in Tcl. The most generic is usually called a *command prefix*, and if the variable *f* holds such a function, then the way to perform the function application *f*(*x*) would be

```
{*}$f $x
```

where `{*}` is the expansion prefix (new in Tcl 8.5). The command prefix in the above example is `apply {x {expr {$x*$x}}}` Command names can be bound to command prefixes by means of the `interp alias` command. Command prefixes support currying. Command prefixes are very common in Tcl APIs.

The other candidate for "function" in Tcl is usually called a *lambda*, and appears as the `{x {expr {$x*$x}}}` part of the above example. This is the part which caches the compiled form of the anonymous function, but it can only be invoked by being passed to the `apply` command. Lambdas do not support currying, unless paired with an `apply` to form a command prefix. Lambdas are rare in Tcl APIs.

Visual Basic .NET

Visual Basic .NET 2008 introduced anonymous functions through the lambda form. Combined with implicit typing, VB provides an economical syntax for anonymous functions. As with Python, in VB.NET, anonymous functions must be defined on one line; they cannot be compound statements. Further, an anonymous function in VB.NET must truly be a VB.NET `Function` - it must return a value.

```
Dim foo = Function(x) x * x
Console.WriteLine(foo(10))
```

Visual Basic.NET 2010 added support for multiline lambda expressions and anonymous functions without a return value. For example, a function for use in a Thread.

```
Dim t As New System.Threading.Thread(Sub ()
                                         For n As Integer = 0 To 10   'Count to 10
                                             Console.WriteLine(n)     'Print each number
                                         Next
                                     End Sub
                                     )
t.Start()
```

Z Shell

The Z Shell has two variants of anonymous function definitions, built after the Bourne and Korn shell variants of regular function definitions:

```
() { echo There are $# txt files in the current directory; } *.txt

function { echo There are $# txt files in the current directory; } *.txt
```

See also

- First-class function

References

1. "Higher order functions" (http://learnyouahaskell.com/higher-order-functions). learnyouahaskell.com. Retrieved 3 December 2014.
2. Fernandez, Maribel (2009), *Models of Computation: An Introduction to Computability Theory* (https://books.google.com/books?id=FPFsnzzebhQC&pg=PA33), Undergraduate Topics in Computer Science, Springer Science & Business Media, p. 33, ISBN 9781848824348, "The Lambda calculus ... was introduced by Alonzo Church in the 1930s as a precise notation for a theory of anonymous functions"
3. "Bash lambda" (https://github.com/spencertipping/bash-lambda). 2019-03-08.
4. "Closure support" (http://www.getrailo.org/index.cfm/whats-up/railo-40-beta-released/features/closures/).
5. "Whats new in ColdFusion 10" (https://learn.adobe.com/wiki/display/coldfusionen/Whats+new+in+ColdFusion+10).
6. "Managed COBOL Reference" (http://documentation.microfocus.com/help/topic/com.microfocus.eclipse.infocenter.visualcobol.vs/GUID-DA75663F-6357-4064-8112-C87E7457DE51.html). *Micro Focus Documentation*. Micro Focus. Retrieved 25 February 2014.
7. "Quotations - Factor Documentation" (http://docs.factorcode.org/content/article-quotations.html). Retrieved 26 December 2015. "A quotation is an anonymous function (a value denoting a snippet of code) which can be used as a value and called using the Fundamental combinators."
8. "Gosu Documentation" (http://gosu-lang.org/doc/pdf/gosuref.pdf) (PDF). Retrieved 4 March 2013.
9. "Groovy Documentation" (http://groovy.codehaus.org/Closures). Retrieved 29 May 2012.
10. Järvi, Jaakko; Powell, Gary (n.d.). "Chapter 16. Boost.Lambda" (http://www.boost.org/doc/libs/1_57_0/doc/html/lambda.html). *Boost Documentation*. Boost. Retrieved December 22, 2014.
11. C# 4.0 Language Specification, section 5.3.3.29 (http://www.microsoft.com/download/en/details.aspx?id=7029)
12. "What's New in JDK 8" (http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html).

13. *The Java Tutorials: Lambda Expressions* (http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html), docs.oracle.com
14. "Programming in Lua - More about Functions" (http://www.lua.org/pil/6.html). Archived (https://web.archive.org/web/20080514220940/http://www.lua.org/pil/6.html) from the original on 14 May 2008. Retrieved 2008-04-25.
15. http://php.net/create_function the top of the page indicates this with "(PHP 4 >= 4.0.1, PHP 5)"
16. "PHP: rfc:closures" (http://wiki.php.net/rfc/closures).
17. "Anonymous Predicates" (http://wiki.visual-prolog.com/index.php?title=Language_Reference/Terms#Anonymous_Predicates). in Visual Prolog Language Reference
18. Sosinski, Robert (2008-12-21). "Understanding Ruby Blocks, Procs and Lambdas" (https://web.archive.org/web/20140531123646/http://www.reactive.io/tips/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/). Reactive.IO. Archived from the original (http://www.reactive.io/tips/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/) on 2014-05-31. Retrieved 2014-05-30.
19. "Anonymous Function Syntax - Scala Documentation" (http://www.scala-lang.org/node/133).
20. "The Swift Programming Language (Swift 3.0.1): Closures" (https://developer.apple.com/library/prerelease/ios/documentation/swift/conceptual/swift_programming_language/Closures.html).
21. *apply manual page* (http://www.tcl.tk/man/tcl8.5/TclCmd/apply.htm), retrieved 2012-09-06.

External links

- Anonymous Methods - When Should They Be Used? (http://www.deltics.co.nz/blog/?p=48) (blog about anonymous function in Delphi)
- Compiling Lambda Expressions: Scala vs. Java 8 (http://www.takipiblog.com/2014/01/16/compiling-lambda-expressions-scala-vs-java-8/)
- php anonymous functions (http://webwidetutor.com/php/PHP-Anonymous-Functions-?id=12) php anonymous functions
- Lambda functions in various programming languages (http://dobegin.com/lambda-functions-everywhere/)