

Closure (computer programming)

From Wikipedia, the free encyclopedia

In programming languages, a **closure** (also **lexical closure** or **function closure**) is a function or reference to a function together with a *referencing environment*—a table storing a reference to each of the non-local variables (also called free variables or upvalues) of that function.^[1] A closure—unlike a plain function pointer—allows a function to access those non-local variables even when invoked outside its immediate lexical scope.

The concept of closures was developed in the 1960s and was first fully implemented in 1975 as a language feature in the Scheme programming language to support lexically scoped first-class functions. The use of closures is associated with functional programming languages such as Lisp and ML. Traditional imperative languages such as Algol, C and Pascal do not support returning nested functions as results from higher-order functions. Many modern garbage-collected imperative languages support closures, such as Smalltalk (the first object-oriented language to do so),^[2] OCaml, Python, Perl, Ruby, JavaScript, Go, Dart, Scala, Swift, C#, Rust, Julia and C++ since C++11.

Contents

- 1 Example
- 2 History and etymology
- 3 Implementation and theory
- 4 Applications
 - 4.1 First-class functions
 - 4.2 State representation
 - 4.3 Other uses
- 5 Differences in semantics
 - 5.1 Lexical environment
 - 5.2 Closure leaving
- 6 Closure-like constructs
 - 6.1 Callbacks (C)
 - 6.2 Local classes and Lambda functions (Java)
 - 6.3 Blocks (C, C++, Objective-C 2.0)
 - 6.4 Delegates (C#, D)
 - 6.5 Function objects (C++)
 - 6.6 Inline agents (Eiffel)
- 7 See also
- 8 References
- 9 External links

Example

The following fragment of Python code defines a function `start` with a local variable `x` and a nested function `increment`. This nested function `increment` has access to `x`, because `x` and the function have the same local scope. The function `start` returns a closure containing a reference to the function `increment`, which adds the `y` value to the `x` value.

```
def start(x):
    def increment(y):
        return x + y
    return increment
```

The closures returned by `start` can be assigned to variables like `first_inc` and `second_inc`. Invoking `increment` through the closures returns the results below:

```
first_inc = start(0)
second_inc = start(8)

first_inc(3)    # returns 3
second_inc(3)  # returns 11

# The x value remains the same for new calls to the function:
first_inc(1)    # returns 1
second_inc(2)  # returns 10
```

History and etymology

Peter J. Landin defined the term *closure* in 1964 as having an *environment part* and a *control part* as used by his SECD machine for evaluating expressions.^[3] Joel Moses credits Landin with introducing the term *closure* to refer to a lambda expression whose open bindings (free variables) have been closed by (or bound in) the lexical environment, resulting in a *closed expression*, or closure.^{[4][5]} This usage was subsequently adopted by Sussman and Steele when they defined Scheme in 1975,^[6] and became widespread.

The term *closure* is often mistakenly used to mean anonymous function. This is probably because many programmers learn about both concepts at the same time, in the form of small helper functions that are anonymous closures. An anonymous function is a function literal without a name, while a closure is an instance of a function, a value, whose non-local variables have been bound either to values or to storage locations (depending on the language; see the lexical environment section below).

Implementation and theory

Closures are typically implemented with a special data structure that contains a pointer to the function code, plus a representation of the function's lexical environment (i.e., the set of available variables) at the time when the closure was created. The referencing environment binds the nonlocal names to the corresponding variables in scope at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime

of the closure itself. When the closure is *entered* at a later time, possibly from a different scope, the function is executed with its non-local variables referring to the ones captured by the closure.

A language implementation cannot easily support full closures if its run-time memory model allocates all local variables on a linear stack. In such languages, a function's local variables are deallocated when the function returns. However, a closure requires that the free variables it references survive the enclosing function's execution. Therefore, those variables must be allocated so that they persist until no longer needed. This explains why, typically, languages that natively support closures also use garbage collection. The alternative is for the language to accept that certain use cases will lead to undefined behaviour, as in lambda expressions in C++11.^[7] The Funarg problem (or "functional argument" problem) describes the difficulty of implementing functions as first class objects in a stack-based programming language such as C or C++. Similarly in D version 1, it is assumed that the programmer knows what to do with delegates and local variables, as their references will be invalid after return from its definition scope (local variables are on the stack) - this still permits many useful functional patterns, but for complex cases needs explicit heap allocation for variables. D version 2 solved this by detecting which variables must be stored on the heap, and performs automatic allocation. Because D uses garbage collection, in both versions, there is no need to track usage of variables as they are passed.

In strict functional languages with immutable data (*e.g.* Erlang), it is very easy to implement automatic memory management (garbage collection), as there are no possible cycles in variables' references. For example in Erlang, all arguments and variables are allocated on the heap, but references to them are additionally stored on the stack. After a function returns, references are still valid. Heap cleaning is done by incremental garbage collector.

In ML, local variables are allocated on a linear stack. When a closure is created, it copies the values of those variables that are needed by the closure into the closure's data structure.

Scheme, which has an ALGOL-like lexical scope system with dynamic variables and garbage collection, lacks a stack programming model and does not suffer from the limitations of stack-based languages. Closures are expressed naturally in Scheme. The lambda form encloses the code and the free variables of its environment, persists within the program as long as it can possibly be accessed, and can be used as freely as any other Scheme expression.

Closures are closely related to Actors in the Actor model of concurrent computation where the values in the function's lexical environment are called *acquaintances*. An important issue for closures in concurrent programming languages is whether the variables in a closure can be updated and, if so, how these updates can be synchronized. Actors provide one solution.^[8]

Closures are closely related to function objects; the transformation from the former to the latter is known as defunctionalization or lambda lifting.

Applications

Closures are used to implement continuation-passing style, and in this manner, hide state. Constructs such as objects and control structures can thus be implemented with closures. In some languages, a closure may occur when a function is defined within another function, and the inner function refers to local variables of the outer function. At run-time, when the outer function executes, a closure is formed, consisting of the inner function's code and references (the upvalues) to any variables of the outer function required by the closure.

First-class functions

Closures typically appear in languages in which functions are first-class values—in other words, such languages allow functions to be passed as arguments, returned from function calls, bound to variable names, etc., just like simpler types such as strings and integers. For example, consider the following Scheme function:

```
; Return a list of all books with at least THRESHOLD copies sold.
(define (best-selling-books threshold)
  (filter
   (lambda (book)
     (>= (book-sales book) threshold))
   book-list))
```

In this example, the lambda expression `(lambda (book) (>= (book-sales book) threshold))` appears within the function `best-selling-books`. When the lambda expression is evaluated, Scheme creates a closure consisting of the code for the lambda expression and a reference to the `threshold` variable, which is a free variable inside the lambda expression.

The closure is then passed to the `filter` function, which calls it repeatedly to determine which books are to be added to the result list and which are to be discarded. Because the closure itself has a reference to `threshold`, it can use that variable each time `filter` calls it. The function `filter` itself might be defined in a completely separate file.

Here is the same example rewritten in JavaScript, another popular language with support for closures:

```
// Return a list of all books with at least 'threshold' copies sold.
function bestSellingBooks(threshold) {
  return bookList.filter(
    function (book) { return book.sales >= threshold; }
  );
}
```

The `function` keyword is used here instead of `lambda`, and an `Array.filter` method^[9] instead of a global `filter` function, but otherwise the structure and the effect of the code are the same.

A function may create a closure and return it, as in the following example:

```
// Return a function that approximates the derivative of f
// using an interval of dx, which should be appropriately small.
function derivative(f, dx) {
  return function (x) {
    return (f(x + dx) - f(x)) / dx;
  };
}
```

Because the closure in this case outlives the scope of the function that creates it, the variables `f` and `dx` live on after the function `derivative` returns. In languages without closures, the lifetime of a local variable coincides with the execution of the scope where that variable is declared. In languages with closures, variables must

continue to exist as long as any existing closures have references to them. This is most commonly implemented using some form of garbage collection.

State representation

A closure can be used to associate a function with a set of "private" variables, which persist over several invocations of the function. The scope of the variable encompasses only the closed-over function, so it cannot be accessed from other program code.

In stateful languages, closures can thus be used to implement paradigms for state representation and information hiding, since the closure's upvalues (its closed-over variables) are of indefinite extent, so a value established in one invocation remains available in the next. Closures used in this way no longer have referential transparency, and are thus no longer pure functions; nevertheless, they are commonly used in impure functional languages such as Scheme.

Other uses

Closures have many uses:

- Because closures delay evaluation—i.e., they do not "do" anything until they are called—they can be used to define control structures. For example, all Smalltalk's standard control structures, including branches (if/then/else) and loops (while and for), are defined using objects whose methods accept closures. Users can easily define their own control structures also.
- In languages that allow assignment, multiple functions can be produced that close over the same environment, enabling them to communicate privately by altering that environment. In Scheme:

```
(define foo #f)
(define bar #f)

(let ((secret-message "none"))
  (set! foo (lambda (msg) (set! secret-message msg)))
  (set! bar (lambda () secret-message)))

(display (bar)) ; prints "none"
(newline)
(foo "meet me by the docks at midnight")
(display (bar)) ; prints "meet me by the docks at midnight"
```

- Closures can be used to implement object systems.^[10]

Note: Some speakers call any data structure that binds a lexical environment a closure, but the term usually refers specifically to functions.

Differences in semantics

Lexical environment

As different languages do not always have a common definition of the lexical environment, their definitions of closure may vary also. The commonly held minimalist definition of the lexical environment defines it as a set of all bindings of variables in the scope, and that is also what closures in any language have to capture. However the meaning of a variable binding also differs. In imperative languages, variables bind to relative locations in memory that can store values. Although the relative location of a binding does not change at runtime, the value in the bound location can. In such languages, since closure captures the binding, any operation on the variable, whether done from the closure or not, are performed on the same relative memory location. This is often called capturing the variable "by reference". Here is an example illustrating the concept in ECMAScript, which is one such language:

```
// ECMAScript
var f, g;
function foo() {
  var x;
  f = function() { return ++x; };
  g = function() { return --x; };
  x = 1;
  alert('inside foo, call to f(): ' + f()); // "2"
}
foo();
alert('call to g(): ' + g()); // "1"
alert('call to f(): ' + f()); // "2"
```

Note how function `foo` and the closures referred to by variables `f` and `g` all use the same relative memory location signified by local variable `x`.

On the other hand, many functional languages, such as ML, bind variables directly to values. In this case, since there is no way to change the value of the variable once it is bound, there is no need to share the state between closures—they just use the same values. This is often called capturing the variable "by value". Java's local and anonymous classes also fall into this category—they require captured local variables to be `final`, which also means there is no need to share state.

Some languages allow you to choose between capturing the value of a variable or its location. For example, in C++11 and PHP, captured variables are either declared with `&`, which means captured by reference, or without, which means captured by value.

Yet another subset, lazy functional languages such as Haskell, bind variables to results of future computations rather than values. Consider this example in Haskell:

```
-- Haskell
foo :: Fractional a => a -> a -> a -> (a -> a)
foo x y = (\z -> z + r)
          where r = x / y

f :: Fractional a => a -> a
f = foo 1 0

main = print (f 123)
```

The binding of `r` captured by the closure defined within function `foo` is to the computation (x / y) - which in this case results in division by zero. However, since it is the computation that is captured, and not the value, the

error only manifests itself when the closure is invoked, and actually attempts to use the captured binding.

Closure leaving

Yet more differences manifest themselves in the behavior of other lexically scoped constructs, such as `return`, `break` and `continue` statements. Such constructs can, in general, be considered in terms of invoking an escape continuation established by an enclosing control statement (in case of `break` and `continue`, such interpretation requires looping constructs to be considered in terms of recursive function calls). In some languages, such as ECMAScript, `return` refers to the continuation established by the closure lexically innermost with respect to the statement—thus, a `return` within a closure transfers control to the code that called it. However in Smalltalk, the superficially similar operator `^` invokes the escape continuation established for the method invocation, ignoring the escape continuations of any intervening nested closures. The escape continuation of a particular closure can only be invoked in Smalltalk implicitly by reaching the end of the closure's code. The following examples in ECMAScript and Smalltalk highlight the difference:

```
"Smalltalk"
foo
| xs |
xs := #(1 2 3 4).
xs do: [:x | ^x].
^0
bar
Transcript show: (self foo printString) "prints 1"
```

```
// ECMAScript
function foo() {
  var xs = [1, 2, 3, 4];
  xs.forEach(function (x) { return x; });
  return 0;
}
alert(foo()); // prints 0
```

The above code snippets will behave differently because the Smalltalk `^` operator and the JavaScript `return` operator are not analogous. In the ECMAScript example, `return x` will leave the inner closure to begin a new iteration of the `forEach` loop, whereas in the Smalltalk example, `^x` will abort the loop and return from the method `foo`.

Common Lisp provides a construct that can express either of the above actions: Lisp `(return-from foo x)` behaves as Smalltalk `^x`, while Lisp `(return-from nil x)` behaves as JavaScript `return x`. Hence, Smalltalk makes it possible for a captured escape continuation to outlive the extent in which it can be successfully invoked. Consider:

```
"Smalltalk"
foo
^[:x | ^x ]
bar
| f |
f := self foo.
f value: 123 "error!"
```

When the closure returned by the method `foo` is invoked, it attempts to return a value from the invocation of `foo` that created the closure. Since that call has already returned and the Smalltalk method invocation model does not follow the spaghetti stack discipline to allow multiple returns, this operation results in an error.

Some languages, such as Ruby, allow the programmer to choose the way `return` is captured. An example in Ruby:

```
# Ruby
# Closure using a Proc
def foo
  f = Proc.new { return "return from foo from inside proc" }
  f.call # control leaves foo here
  return "return from foo"
end

# Closure using a lambda
def bar
  f = lambda { return "return from lambda" }
  f.call # control does not leave bar here
  return "return from bar"
end

puts foo # prints "return from foo from inside proc"
puts bar # prints "return from bar"
```

Both `Proc.new` and `lambda` in this example are ways to create a closure, but semantics of the closures thus created are different with respect to the `return` statement.

In Scheme, definition and scope of the `return` control statement is explicit (and only arbitrarily named 'return' for the sake of the example). The following is a direct translation of the Ruby sample.

```
; Scheme
(define call/cc call-with-current-continuation)

(define (foo)
  (call/cc
   (lambda (return)
     (define (f) (return "return from foo from inside proc"))
     (f) ; control leaves foo here
     (return "return from foo")))))

(define (bar)
  (call/cc
   (lambda (return)
     (define (f) (call/cc (lambda (return) (return "return from lambda"))))
     (f) ; control does not leave bar here
     (return "return from bar")))))

(display (foo)) ; prints "return from foo from inside proc"
(newline)
(display (bar)) ; prints "return from bar"
```

Closure-like constructs

Features of some languages simulate some features of closures. Language features include some object-oriented techniques, for example in Java, C++, Objective-C, C#, D.

Callbacks (C)

In C, libraries that support callbacks sometimes allow a callback to be registered using two values: a function pointer and a separate `void*` pointer to arbitrary data of the user's choice. Each time the library executes the callback function, it passes in the data pointer. This allows the callback to maintain state and to refer to information captured at the time it was registered. The idiom is similar to closures in functionality, but not in syntax.

Local classes and Lambda functions (Java)

Java allows classes to be defined inside methods. These are called *local classes*. When such classes are not named, they are known as *anonymous classes* (or anonymous *inner* classes). A local class (either named or anonymous) may refer to names in lexically enclosing classes, or read-only variables (marked as `final`) in the lexically enclosing method.

```
class CalculationWindow extends JFrame {
    private volatile int result;
    ...
    public void calculateInSeparateThread(final URI uri) {
        // The expression "new Runnable() { ... }" is an anonymous class implementing the 'Runnable' interface.
        new Thread(
            new Runnable() {
                void run() {
                    // It can read final local variables:
                    calculate(uri);
                    // It can access private fields of the enclosing class:
                    result = result + 10;
                }
            }
        ).start();
    }
}
```

The capturing of `final` variables allows you to capture variables by value. Even if the variable you want to capture is non-`final`, you can always copy it to a temporary `final` variable just before the class.

Capturing of variables by reference can be emulated by using a `final` reference to a mutable container, for example, a single-element array. The local class will not be able to change the value of the container reference itself, but it will be able to change the contents of the container.

With the advent of Java 8's lambda expressions,^[11] the closure allows the above code to be executed as:

```
class CalculationWindow extends JFrame {
    private volatile int result;
    ...
    public void calculateInSeparateThread(final URI uri) {
```

```
// the code () -> { /* code */ } is a closure
new Thread(() -> {
    calculate(uri);
    result = result + 10;
}).start();
}
```

Local classes are one of the types of inner class that are declared within the body of a method. Java also supports inner classes that are declared as *non-static members* of an enclosing class.^[12] They are normally referred to just as "inner classes".^[13] These are defined in the body of the enclosing class and have full access to instance variables of the enclosing class. Due to their binding to these instance variables, an inner class may only be instantiated with an explicit binding to an instance of the enclosing class using a special syntax.^[14]

```
public class EnclosingClass {
    /* Define the inner class */
    public class InnerClass {
        public int incrementAndReturnCounter() {
            return counter++;
        }
    }

    private int counter;

    {
        counter = 0;
    }

    public int getCounter() {
        return counter;
    }

    public static void main(String[] args) {
        EnclosingClass enclosingClassInstance = new EnclosingClass();
        /* Instantiate the inner class, with binding to the instance */
        EnclosingClass.InnerClass innerClassInstance =
            enclosingClassInstance.new InnerClass();

        for(int i = enclosingClassInstance.getCounter(); (i =
            innerClassInstance.incrementAndReturnCounter()) < 10;) {
            System.out.println(i);
        }
    }
}
```

Upon execution, this will print the integers from 0 to 9. Beware to not confuse this type of class with the nested class, which is declared in the same way with an accompanied usage of the "static" modifier; those have not the desired effect but are instead just classes with no special binding defined in an enclosing class.

There have been a number of proposals for adding more fully featured closures to Java.^[15]

Blocks (C, C++, Objective-C 2.0)

Apple introduced **Blocks**, a form of closure, as a nonstandard extension into C, C++, Objective-C 2.0 and in Mac OS X 10.6 "Snow Leopard" and iOS 4.0.

Pointers to block and block literals are marked with ^. Normal local variables are captured by value when the

block is created, and are read-only inside the block. Variables to be captured by reference are marked with `__block`. Blocks that need to persist outside of the scope they are created in may need to be copied.^{[16][17]}

```
typedef int (^IntBlock)();

IntBlock downCounter(int start) {
    __block int i = start;
    return [ [ ^int() {
        return i--;
    } copy] autorelease];
}

IntBlock f = downCounter(5);
NSLog(@"%d", f());
NSLog(@"%d", f());
NSLog(@"%d", f());
```

Delegates (C#, D)

C# anonymous methods and lambda expressions support closure:

```
var data = new[] {1, 2, 3, 4};
var multiplier = 2;
var result = data.Select(x => x * multiplier);
```

Closures are implemented by delegates in D.

```
auto test1() {
    int a = 7;
    return delegate() { return a + 3; }; // anonymous delegate construction
}

auto test2() {
    int a = 20;
    int foo() { return a + 5; } // inner function
    return &foo; // other way to construct delegate
}

void bar() {
    auto dg = test1();
    dg(); // =10 // ok, test1.a is in a closure and still exists

    dg = test2();
    dg(); // =25 // ok, test2.a is in a closure and still exists
}
```

D version 1, has limited closure support. For example, the above code will not work correctly, because the variable `a` is on the stack, and after returning from `test()`, it is no longer valid to use it (most probably calling `foo` via `dg()`, will return a 'random' integer). This can be solved by explicitly allocating the variable `'a'` on heap, or using structs or class to store all needed closed variables and construct a delegate from a method implementing the same code. Closures can be passed to other functions, as long as they are only used while the referenced

values are still valid (for example calling another function with a closure as a callback parameter), and are useful for writing generic data processing code, so this limitation, in practice, is often not an issue.

This limitation was fixed in D version 2 - the variable 'a' will be automatically allocated on the heap because it is used in the inner function, and a delegate of that function is allowed to escape the current scope (via assignment to `dg` or `return`). Any other local variables (or arguments) that are not referenced by delegates or that are only referenced by delegates that don't escape the current scope, remain on the stack, which is simpler and faster than heap allocation. The same is true for inner's class methods that references a function's variables.

Function objects (C++)

C++ allows defining function objects by overloading `operator()`. These objects behave somewhat like functions in a functional programming language. They may be created at runtime and may contain state, but they do not implicitly capture local variables as closures do. As of the 2011 revision, the C++ language also supports closures, which are a type of function object constructed automatically from a special language construct called *lambda-expression*. A C++ closure may capture its context either by storing copies of the accessed variables as members of the closure object or by reference. In the latter case, if the closure object escapes the scope of a referenced object, invoking its `operator()` causes undefined behavior since C++ closures do not extend the lifetime of their context.

```
void foo(string myname) {
    int y;
    vector<string> n;
    // ...
    auto i = std::find_if(n.begin(), n.end(),
        // this is the lambda expression:
        [&](const string& s) { return s != myname && s.size() > y; }
    );
    // 'i' is now either 'n.end()' or points to the first string in 'n'
    // which is not equal to 'myname' and whose length is greater than 'y'
}
```

Inline agents (Eiffel)

Eiffel includes **inline agents** defining closures. An inline agent is an object representing a routine, defined by giving the code of the routine in-line. For example, in

```
ok_button.click_event.subscribe (
    agent (x, y: INTEGER) do
        map.country_at_coordinates (x, y).display
    end
)
```

the argument to `subscribe` is an agent, representing a procedure with two arguments; the procedure finds the country at the corresponding coordinates and displays it. The whole agent is "subscribed" to the event type `click_event` for a certain button, so that whenever an instance of the event type occurs on that button — because a user has clicked the button — the procedure will be executed with the mouse coordinates being passed as arguments for `x` and `y`.

The main limitation of Eiffel agents, which distinguishes them from true closures, is that they cannot reference local variables from the enclosing scope. Only `current` (a reference to current object, analogous to `this` in Java), its features, and arguments of the agent itself can be accessed from within the agent body. This limitation is worked around by providing additional closed operands to the agent.

See also

- Anonymous function
- Blocks (C language extension)
- Command pattern
- Continuation
- Currying
- Funarg problem
- Lambda calculus
- Lazy evaluation
- Partial application
- Spaghetti stack
- Syntactic closure
- Value-level programming

References

- ↑ Sussman and Steele. "Scheme: An interpreter for extended lambda calculus". "... a data structure containing a lambda expression, and an environment to be used when that lambda expression is applied to arguments." (Wikisource)
- ↑ Closures in Java (http://www.jugpadova.it/files/Closures_in_Java.pdf)
- ↑ P. J. Landin (1964), *The mechanical evaluation of expressions*
- ↑ Joel Moses (June 1970), *The Function of FUNCTION in LISP, or Why the FUNARG Problem Should Be Called the Environment Problem* (<http://dspace.mit.edu/handle/1721.1/5854>) (PDF), AI Memo 199, retrieved 2009-10-27, "A useful metaphor for the difference between FUNCTION and QUOTE in LISP is to think of QUOTE as a porous or an open covering of the function since free variables escape to the current environment. FUNCTION acts as a closed or nonporous covering (hence the term "closure" used by Landin). Thus we talk of "open" Lambda expressions (functions in LISP are usually Lambda expressions) and "closed" Lambda expressions. [...] My interest in the environment problem began while Landin, who had a deep understanding of the problem, visited MIT during 1966-67. I then realized the correspondence between the FUNARG lists which are the results of the

- evaluation of "closed" Lambda expressions in LISP and ISWIM's Lambda Closures."
5. ^ Åke Wikström (1987). *Functional Programming using Standard ML*. ISBN 0-13-331968-7. "The reason it is called a "closure" is that an expression containing free variables is called an "open" expression, and by associating to it the bindings of its free variables, you close it."
 6. ^ Gerald Jay Sussman and Guy L. Steele, Jr. (December 1975), *Scheme: An Interpreter for the Extended Lambda Calculus*, AI Memo 349
 7. ^ *Lambda Expressions and Closures* (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf>) C++ Standards Committee. 29 February 2008.
 8. ^ *Foundations of Actor Semantics* (<https://dspace.mit.edu/handle/1721.1/6935>) Will Clinger. MIT Mathematics Doctoral Dissertation. June 1981.
 9. ^ "array.filter" (https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Array/filter) . *Mozilla Developer Center*. 10 January 2010. Retrieved 2010-02-09.
 10. ^ "Re: FP, OO and relations. Does anyone trump the others?" (<http://okmij.org/ftp/Scheme/oo-in-fp.txt>) . 29 December 1999. Retrieved 2008-12-23.
 11. ^ "Lambda Expressions (The Java Tutorials)" (<http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>) .
 12. ^ "Nested, Inner, Member, and Top-Level Classes" (https://blogs.oracle.com/darcy/entry/nested_inner_member_and_top) .
 13. ^ "Inner Class Example (The Java Tutorials > Learning the Java Language > Classes and Objects)" (<http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html>) .
 14. ^ "Nested Classes (The Java Tutorials > Learning the Java Language > Classes and Objects)" (<http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html>) .
 15. ^ Closures (Lambda Expressions) for the Java Programming Language (<http://www.javac.info/>) . Javac.info (2008-08-11). Retrieved on 2013-07-31.
 16. ^ Apple Inc. "Blocks Programming Topics" (http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html) . Retrieved 2011-03-08.
 17. ^ Joachim Bengtsson (7 July 2010). "Programming with C Blocks On Apple Devices" (<http://thirdcog.eu/pwcblocks/>) . Retrieved 2010-09-18.

External links

- The Original "Lambda Papers" (<http://library.readscheme.org/page1.html>) : A classic series of papers by Guy Steele and Gerald Sussman discussing, among other things, the versatility of closures in the context of Scheme (where they appear as *lambda expressions*).
- Neal Gafter (2007-01-28). "A Definition of Closures" (<http://gafter.blogspot.com/2007/01/definition-of-closures.html>) .
- Gilad Bracha, Neal Gafter, James Gosling, Peter von der Ahé. "Closures for the Java Programming Language (v0.5)" (<http://www.javac.info/closures-v05.html>) .
- Closures (<http://martinfowler.com/bliki/Closure.html>) : An article about closures in dynamically typed imperative languages, by Martin Fowler.
- Collection closure methods (<http://martinfowler.com/bliki/CollectionClosureMethod.html>) : An example of a technical domain where using closures is convenient, by Martin Fowler.

Javascript

- What are closures (<http://blogs.msdn.com/kartikb/archive/2009/02/08/closures.aspx>) : A post on closures in Javascript.
- Javascript Closures for Dummies (http://web.archive.org/web/20080209105120/http://blog.morrisjohns.com/javascript_closures_for_dummies) : An article teaching closures in Javascript by examples.

Java and .NET

- The beauty of closures (<http://www.developerfusion.com/article/8251/the-beauty-of-closures/>) : An article about using closures in Java and .NET

Delphi

- Nick Hodges, "Delphi 2009 Reviewers Guide (<http://dn.codegear.com/article/38757>)", October 2008, *CodeGear Developer Network*, CodeGear.
- Craig Stuntz, "Understanding Anonymous Methods (<http://blogs.teamb.com/craigstuntz/2008/08/04/37828/>)", October 2008
- Dr. Bob, "Delphi 2009 Anonymous Methods (<http://www.drbob42.com/examines/examinA5.htm>)"

Ruby

- Robert Sosinski, "Understanding Ruby Blocks, Procs and Lambdas (<http://www.robertsosinski.com/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/>)", December 2008

Retrieved from "[http://en.wikipedia.org/w/index.php?title=Closure_\(computer_programming\)&oldid=613456018](http://en.wikipedia.org/w/index.php?title=Closure_(computer_programming)&oldid=613456018)"

Categories: Programming language concepts | Implementation of functional programming languages

| Subroutines | Articles with example Scheme code | Articles with example JavaScript code

| Articles with example Eiffel code | Articles with example C Sharp code | Articles with example D code

| Articles with example Objective-C code | Articles with example Ruby code

| Articles with example Smalltalk code

-
- This page was last modified on 18 June 2014 at 17:51.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.