　当function里嵌套function时，内部的function可以访问外部function里的变量。

function foo(x) {  
    var tmp = 3;  
    function bar(y) {  
        alert(x + y + (++tmp));  
    }  
    bar(10);  
}  
foo(2)

　　不管执行多少次，都会alert 16，因为bar能访问foo的参数x，也能访问foo的变量tmp。

　　但，这还不是闭包。当你return的是内部function时，就是一个闭包。内部function会close-over外部function的变量直到内部function结束。

function foo(x) {

    var tmp = 3;  
    return function (y) {  
        alert(x + y + (++tmp));  
    }  
}  
var bar = foo(2); // bar 现在是一个闭包  
bar(10);

　　上面的脚本最终也会alert 16，因为虽然bar不直接处于foo的内部作用域，但bar还是能访问x和tmp。

　　但是，由于tmp仍存在于bar闭包的内部，所以它还是会自加1，而且你每次调用bar时它都会自加1.

　　（考虑到六岁这个限制：我们其实可以建立不止一个闭包方法，比如return它们的数组，也可以把它们设置为全局变量。它们全都指向相同的x和相同的tmp，而不是各自有一份副本。）

　　注：现在来整点儿七岁的内容。

　　上面的x是一个字面值（值传递），和JS里其他的字面值一样，当调用foo时，实参x的值被复制了一份，复制的那一份作为了foo的参数x。

　　那么问题来了，JS里处理object时是用到引用传递的，那么，你调用foo时传递一个object，foo函数return的闭包也会引用最初那个object！

function foo(x) {  
var tmp = 3;  
return function (y) {  
    alert(x + y + tmp);  
    x.memb = x.memb ? x.memb + 1 : 1;  
    alert(x.memb);  
    }  
}  
var age = new Number(2);  
var bar = foo(age); // bar 现在是一个引用了age的闭包  
bar(10);

　　不出我们意料，每次运行bar(10)，x.memb都会自加1。但需要注意的是x每次都指向同一个object变量——age，运行两次bar(10)后，age.memb会变成2.

　　这和HTML对象的内存泄漏有关，呃，不过貌似超出了答题的范围。

　　JohnMerlino 对Ali说：

　　这里有一个不用return关键字的闭包例子：

function closureExample(objID, text, timedelay) {   
    setTimeout(function() {   
        document.getElementById(objID).innerHTML = text;   
    }, timedelay);   
}   
closureExample(‘myDiv’, ‘Closure is created’, 500);

　　深夜1:37 John Pick这样回答：

　　JS里的function能访问它们的：

　　1. 参数

　　2. 局部变量或函数

　　3. 外部变量（环境变量？），包括

3.1 全局变量，包括DOM。

3.2 外部函数的变量或函数。

　　如果一个函数访问了它的外部变量，那么它就是一个闭包。

　　注意，外部函数不是必需的。通过访问外部变量，一个闭包可以维持（keep alive）这些变量。在内部函数和外部函数的例子中，外部函数可以创建局部变量，并且最终退出；但是，如果任何一个或多个内部函数在它退出后却没有退出，那么内部函数就维持了外部函数的局部数据。

　　一个典型的例子就是全局变量的使用。

　　mykhal这样回答：

　　Wikipedia对闭包的定义是这样的：

In computer science, a closure is a function together with a referencing environment for the nonlocal names (free variables) of that function.

　　从技术上来讲，在JS中，每个function都是闭包，因为它总是能访问在它外部定义的数据。

　　Since ****scope-defining construction in Javascript is a function****, not a code block like in many other languages, ****what we usually mean by** closure **in Javascript**** is a ****fuction working with nonlocal variables defined in already executed surrounding function****.

　　闭包经常用于创建含有隐藏数据的函数（但并不总是这样）。

var db = (function() {  
// 创建一个隐藏的object, 这个object持有一些数据  
// 从外部是不能访问这个object的  
var data = {};  
// 创建一个函数, 这个函数提供一些访问data的数据的方法  
return function(key, val) {  
    if (val === undefined) { return data[key] } // get  
    else { return data[key] = val } // set  
    }  
// 我们可以调用这个匿名方法  
// 返回这个内部函数，它是一个闭包  
})();  
  
db('x'); // 返回 undefined  
db('x', 1); // 设置data['x']为1  
db('x'); // 返回 1  
// 我们不可能访问data这个object本身  
// 但是我们可以设置它的成员

维基百科上对闭包的解释就很经典：

在计算机科学中，闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。

Peter J. Landin 在1964年将术语闭包定义为一种包含环境成分和控制成分的实体。

下面是我理解的闭包概念。

先看看数学上的闭包。

(1,5) 是一个区间，但对这个区间做分析、计算什么的，经常会用到1和5这两个不属于这个区间的值，[1,5]就是(1,5)的闭包。

在生活上，我们办事情，找A部门，A部门说，你先得找B部门盖个章，B部门说，你先得找C部门盖个章，C部门说，这个东西不是我们的职权范围…… 踢皮球，这就是非闭包。闭包就是负责到底，你找到A部门，A部门接待的那个人负责到底，他/她去协调B部门和C部门。

在工程上，闭包就是项目经理，负责调度项目所需要的资源。老板、客户有什么事情，直接找项目经理即可，不用再去找其它的人。

在程序语言中，闭包就是一种语法糖，它以很自然的形式，把我们的目的和我们的目的所涉及的资源全给自动打包在一起，以某种自然、尽量不让人误解的方式让人来使用。至于其具体实现，我个人意见，在不影响使用的情况下，不求甚解即可。在很多情况下，需要在一段代码里去访问外部的局部变量，不提供这种语法糖，需要写非常多的代码，有了闭包这个语法糖，就不用写这么多代码，自然而然的就用了。

这样一来，可以把闭包从一个语法机制提升为一种设计原则：

闭包是从用户角度考虑的一种设计概念，它基于对上下文的分析，把龌龊的事情、复杂的事情和外部环境交互的事情都自己做了，留给用户一个很自然的接口。

在这个原则下，函数式语言中，那种所谓的闭包只是一种“闭包”，还有大量的其它类型的“闭包”等待发现和实现。