Blocks

Blocks are Objective-C's anonymous functions. They let you pass arbitrary statements between objects as you would data, which is often more intuitive than referencing functions defined elsewhere. In addition, blocks are implemented as *closures*, making it trivial to capture the surrounding state.

Creating Blocks

Blocks use all the same mechanics as normal functions. You can declare a block variable just like you would declare a function, define the block as though you would implement a function, and then call the block as if it were a function:

```
// main.m
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // Declare the block variable
        double (^distanceFromRateAndTime)(double rate, double time);

        // Create and assign the block
        distanceFromRateAndTime = ^double(double rate, double time) {
            return rate * time;
        };
        // Call the block
        double dx = distanceFromRateAndTime(35, 1.5);

        NSLog(@"A car driving 35 mph will travel "
              @"%.2f miles in 1.5 hours.", dx);
    }
    return 0;
}
```

The caret (`^`) is used to mark the `distanceFromRateAndTime` variable as a block. Like a function declaration, you need to include the return type and parameter types so the compiler can enforce type safety. The `^` behaves in a similar manner to the asterisk before a pointer (e.g., `int *aPointer`) in that it is only required when *declaring* the block, after which you can use it like a normal variable.

The block itself is essentially a function definition—without the function name. The `^double(double rate, double time)` signature begins a block literal that returns a `double` and has two parameters that are also doubles (the return type can be omitted if desired). Arbitrary statements can go inside the curly braces (`{}`), just like a normal function.

After assigning the block literal to the `distanceFromRateAndTime` variable, we can *call* that variable as though it were a function.

Parameterless Blocks

If a block doesn't take any parameters, you can omit the argument list in its entirety. And again, specifying the return type of a block literal is always optional, so you can shorten the notation to `^ { ... }`:

```
double (^randomPercent)(void) = ^ {
    return (double)arc4random() / 4294967295;
};
NSLog(@"Gas tank is %.1f%% full",
      randomPercent() * 100);
```

The built-in `arc4random()` function returns a random 32-bit integer. By dividing by the maximum possible value of `arc4random()` (4294967295), we get a decimal value between 0 and 1.

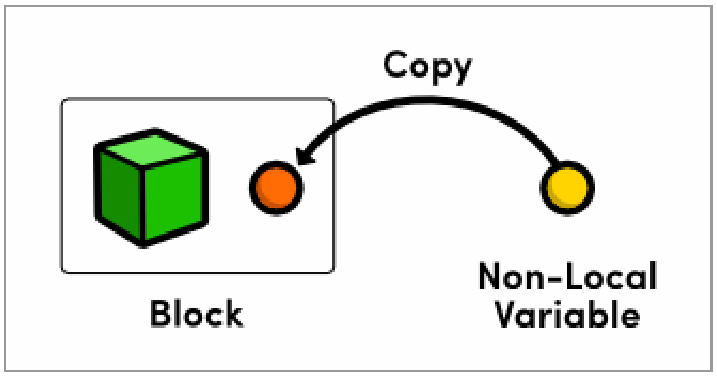
So far, it might seem like blocks are just a complicated way of defining functions. But, the fact that they're implemented as *closures* opens the door to exciting new programming opportunities.

Closures

Inside of a block, you have access to same data as in a normal function: local variables, parameters passed to the block, and global variables/functions. But, blocks are implented as **closures**, which means that you also have access to **non-local variables**. Non-local variables are variables defined in the block's enclosing lexical scope, but outside the block itself. For example, `getFullCarName` can reference the `make` variable defined before the block:

```
NSString *make = @"Honda";
NSString *(^getFullCarName)(NSString *) = ^(NSString *model) {
    return [make stringByAppendingFormat:@" %@", model];
};
NSLog(@"%@", getFullCarName(@"Accord"));    // Honda Accord
```

Non-local variables are copied and stored with the block as `const` variables, which means they are read-only. Trying to assign a new value to the `make` variable from inside the block will throw a compiler error.



Accessing non-local variables as `const` copies

The fact that non-local variables are copied as constants means that a block doesn't just have *access* to non-local variables—it creates a *snapshot* of them. Non-local variables are frozen at whatever value they contain when the block is defined, and the block *always* uses that value, even if the non-local variable changes later on in the program. Watch what happens when we try to change the `make` variable after creating the block:

```
NSString *make = @"Honda";
NSString *(^getFullCarName)(NSString *) = ^(NSString *model) {
    return [make stringByAppendingFormat:@" %@", model];
};
NSLog(@"%@", getFullCarName(@"Accord"));    // Honda Accord

// Try changing the non-local variable (it won't change the block)
make = @"Porsche";
NSLog(@"%@", getFullCarName(@"911 Turbo")); // Honda 911 Turbo
```

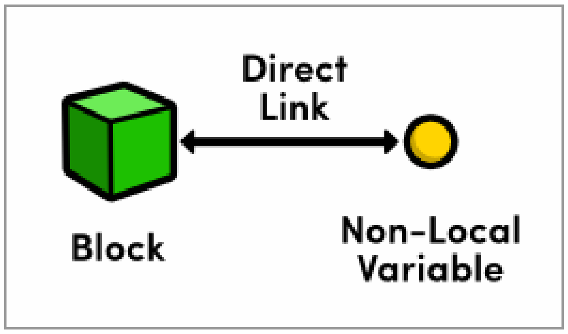
Closures are an incredibly convenient way to work with the surrounding state, as it eliminates the need to pass in extra values as parameters—you simply use non-local variables as if they were defined in the block itself.

Mutable Non-Local Variables

Freezing non-local variables as constant values is a safe default behavior in that it prevents you from accidentally changing them from within the block; however, there are occasions when this is desirable. You can override the `const` copy behavior by declaring a non-local variable with the `__block` storage modifier:

```
__block NSString *make = @"Honda";
```

This tells the block to capture the variable *by reference*, creating a direct link between the `make` variable outside the block and the one inside the block. You can now assign a new value to `make` from outside the block, and it will be reflected in the block, and vice versa.



Accessing non-local variables by reference

Like static local variables in normal functions, the `__block` modifier serves as a “memory” between multiple calls to a block. This makes it possible to use blocks as generators. For example, the following snippet creates a block that “remembers” the value of `i` over subsequent invocations.

```
__block int i = 0;
int (^count)(void) = ^ {
    i += 1;
    return i;
};
NSLog(@"%d", count());    // 1
NSLog(@"%d", count());    // 2
NSLog(@"%d", count());    // 3
```

Blocks as Method Parameters

Storing blocks in variables is occasionally useful, but in the real world, they're more likely to be used as method parameters. They solve the same problem as function pointers, but the fact that they can be defined inline makes the resulting code much easier to read.

For example, the following `Car` interface declares a method that tallies the distance traveled by the car. Instead of forcing the caller to pass a constant speed, it accepts a block that defines the car's speed as a function of time.

```
// Car.h
#import <Foundation/Foundation.h>

@interface Car : NSObject

@property double odometer;

- (void)driveForDuration:(double)duration
       withVariableSpeed:(double (^)(double time))speedFunction
                   steps:(int)numSteps;

@end
```

The data type for the block is `double (^)(double time)`, which states that whatever block the caller passes to the method should return a `double` and accept a single `double` parameter. Note that this is almost the exact same syntax as the block variable declaration discussed at the beginning of this module, but without the variable name.

The implementation can then call the block via `speedFunction`. The following example uses a naïve right-handed Riemann sum to approximate the distance traveled over `duration`. The `steps` argument is used to let the caller determine the precision of the estimate.

```
// Car.m
#import "Car.h"

@implementation Car

@synthesize odometer = _odometer;

- (void)driveForDuration:(double)duration
       withVariableSpeed:(double (^)(double time))speedFunction
                   steps:(int)numSteps {
    double dt = duration / numSteps;
    for (int i=1; i<=numSteps; i++) {
        _odometer += speedFunction(i*dt) * dt;
    }
}

@end
```

As you can see in the `main()` function included below, block literals can be defined *within* a method invocation. While it might take a second to parse the syntax, this is still much more intuitive than creating a dedicated top-level function to define the `withVariableSpeed` parameter.

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *theCar = [[Car alloc] init];

        // Drive for awhile with constant speed of 5.0 m/s
        [theCar driveForDuration:10.0
               withVariableSpeed:^(double time) {
                           return 5.0;
                       } steps:100];
        NSLog(@"The car has now driven %.2f meters", theCar.odometer);

        // Start accelerating at a rate of 1.0 m/s^2
        [theCar driveForDuration:10.0
               withVariableSpeed:^(double time) {
                           return time + 5.0;
                       } steps:100];
        NSLog(@"The car has now driven %.2f meters", theCar.odometer);
    }
    return 0;
}
```

This is a simple example of the versatility of blocks, but the standard frameworks are chock-full of other use cases. `NSArray` lets you sort elements with a block via the `sortedArrayUsingComparator:` method, and `UIView` uses a block to define the final state of an animation via the `animateWithDuration:animations:` method. Blocks are also a powerful tool for concurrent programming.

Defining Block Types

Since the syntax for block data types can quickly clutter up your method declarations, it's often useful to `typedef` common block signatures. For instance, the following code creates a new type called `SpeedFunction` that we can use as a more semantic data type for the `withVariableSpeed` argument.

```
// Car.h
#import <Foundation/Foundation.h>

// Define a new type for the block
typedef double (^SpeedFunction)(double);

@interface Car : NSObject
```

```
@property double odometer;

- (void)driveForDuration:(double)duration
       withVariableSpeed:(SpeedFunction)speedFunction
                   steps:(int)numSteps;

@end
```

Many of the standard Objective-C frameworks also use this technique (e.g., `NSComparator`).

Summary

Blocks provide much the same functionality as C functions, but they are much more intuitive to work with (after you get used to the syntax). The fact that they can be defined inline makes it easy to use them inside of method calls, and since they are closures, it's possible to capture the value of surrounding variables with literally no additional effort.

The next module switches gears a little bit and delves into iOS's and OS X's error-handling capabilities. We'll explore two important classes for representing errors: `NSException` and `NSError`.

Continue to *Exceptions & Errors* ›