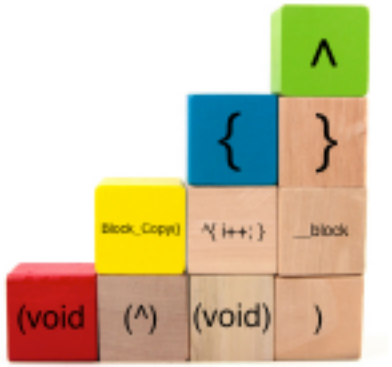
ITP 342
Mobile App Dev



Code Blocks

Working with Blocks

- Blocks are a language-level feature added to C, Objective-C and C++, which allow you to create distinct segments of code that can be passed around to methods or functions as if they were values.
- Blocks are Objective-C objects, which means they can be added to collections like NSArray or NSDictionary.
- They also have the ability to capture values from the enclosing scope, making them similar to closures or lambdas in other programming languages.

Block Syntax

- The syntax to define a block literal uses the caret symbol (^), like this:

```
^{
    NSLog(@"This is a block");
}
```

- As with function and method definitions, the braces indicate the start and end of the block.
- In this example, the block doesn't return any value, and doesn't take any arguments.

Block Syntax

- In the same way that you can use a function pointer to refer to a C function, you can declare a variable to keep track of a block, like this:

```
void (^simpleBlock)(void);
```

- This example declares a variable called simpleBlock to refer to a block:

```
simpleBlock = ^{
   NSLog(@"This is a block");
};
```

- This is just like any other variable assignment, so the statement must be terminated by a semi-colon after the closing brace

Block Syntax

- You can also combine the variable declaration and assignment:

```
void (^simpleBlock)(void) = ^{
   NSLog(@"This is a block");
};
```

- Once you've declared and assigned a block variable, you can use it to invoke the block:

```
simpleBlock();
```

Arguments and Return Values

- Blocks can also take arguments and return values just like methods and functions.
- As an example, consider a variable to refer to a block that returns the result of multiplying two values:

```
double (^multiplyTwoValues)(double, double);
```

- The corresponding block literal might look like this:

```
^ (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

- The return type is inferred from the return statement inside the block.

Arguments and Return Values

- The firstValue and secondValue are used to refer to the values supplied when the block is invoked, just like any function definition.
- If you prefer, you can make the return type explicit by specifying it between the caret and the argument list:

```
^ double (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

Arguments and Return Values

- Once you've declared and defined the block, you can invoke it just like you would a function:

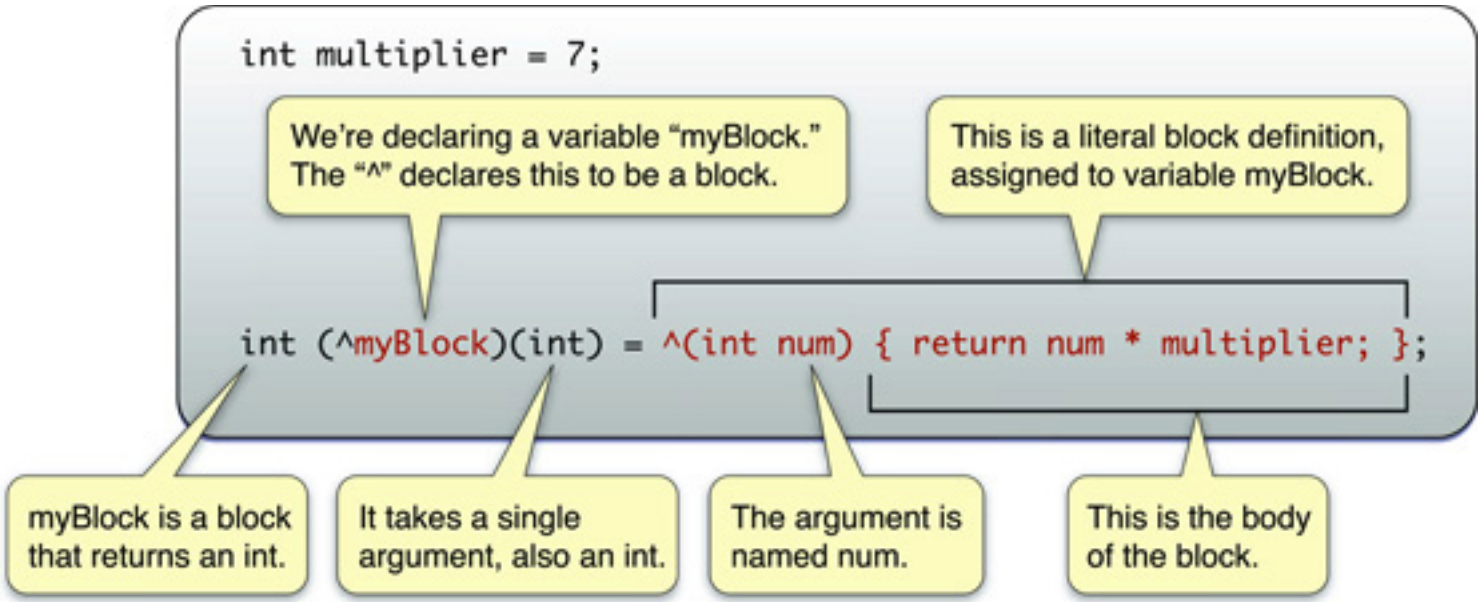
```
double (^multiplyTwoValues)(double, double) =
    ^ double (double firstValue, double secondValue) {
        return firstValue * secondValue;
    };

double result = multiplyTwoValues(2,4);

NSLog(@"The result is %f", result);
```

Declaring a Block

- You use the ^ operator to declare a block variable and to indicate the beginning of a block literal.
- The body of the block itself is contained within {}.



Capture Values from Enclosing Scope

- As well as containing executable code, a block also has the ability to capture state from its enclosing scope.
- If you declare a block literal from within a method, for example, it's possible to capture any of the values accessible within the scope of that method, like this:

```
- (void) testMethod {
    int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
    };

    testBlock();
}
```

- In this example, `anInteger` is declared outside of the block, but the value is captured when the block is defined.

```
Integer is: 42
```

Capture Values from Enclosing Scope

- Only the value is captured, unless you specify otherwise.
- This means that if you change the external value of the variable between the time you define the block and the time it's invoked, like this:

```
int anInteger = 42;

void (^testBlock)(void) = ^{
   NSLog(@"Integer is: %i", anInteger);
};

anInteger = 84;

testBlock();
```

- The value captured by the block is unaffected.
- This means that the log output would still show:

```
Integer is: 42
```

Variables to Share Storage

- If you need to be able to change the value of a captured variable from within a block, you can use the **`__block`** (2 underscores) storage type modifier on the original variable declaration.
- This means that the variable lives in storage that is shared between the lexical scope of the original variable and any blocks declared within that scope.

Variables to Share Storage

```
__block int anInteger = 42;

void (^testBlock)(void) = ^{
   NSLog(@"Integer is: %i", anInteger);
};

anInteger = 84;

testBlock();
```

- Because `anInteger` is declared as a `__block` variable, its storage is shared with the block declaration.
- This means that the log output would now show:

```
Integer is: 84
```

Variables to Share Storage

- It also means that the block can modify the original value, like this:

```
__block int anInteger = 42;

void (^testBlock)(void) = ^{
   NSLog(@"Integer is: %i", anInteger);
   anInteger = 100;
};

testBlock();
NSLog(@"Value of original variable is now: %i", anInteger);
```

- This means that the log output would now show:

```
Integer is: 42
Value of original variable is now: 100
```

Pass Blocks as Arguments

- It's common to pass blocks to functions or methods for invocation elsewhere.
- Blocks are also used for callbacks, defining the code to be executed when a task completes.
  - You can define the callback behavior at the time you initiate the task.
- Several methods in the Cocoa frameworks take **a block as an argument**, typically either to perform an operation on a collection of objects, or to use as a callback after an operation has finished.

Block Should Be Last

- It's best practice to use only one block argument to a method.
- If the method also needs other non-block arguments, the block should come last:

```
- (void)beginTaskWithName: (NSString *) name
   completion: (void(^)(void)) callback;
```

- This makes the method call easier to read when specifying the block inline, like this:

```
[self beginTaskWithName: @"MyTask"
   completion: ^{
      NSLog(@"The task is complete");
   }
];
```

Resources

- Apple's Developer Site:
  - https://developer.apple.com/library/ios/DOCUMENTATION/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html
- The Pragmatic Studio's Using Blocks:
  - https://pragmaticstudio.com/blog/2010/7/28/ios4-blocks-1