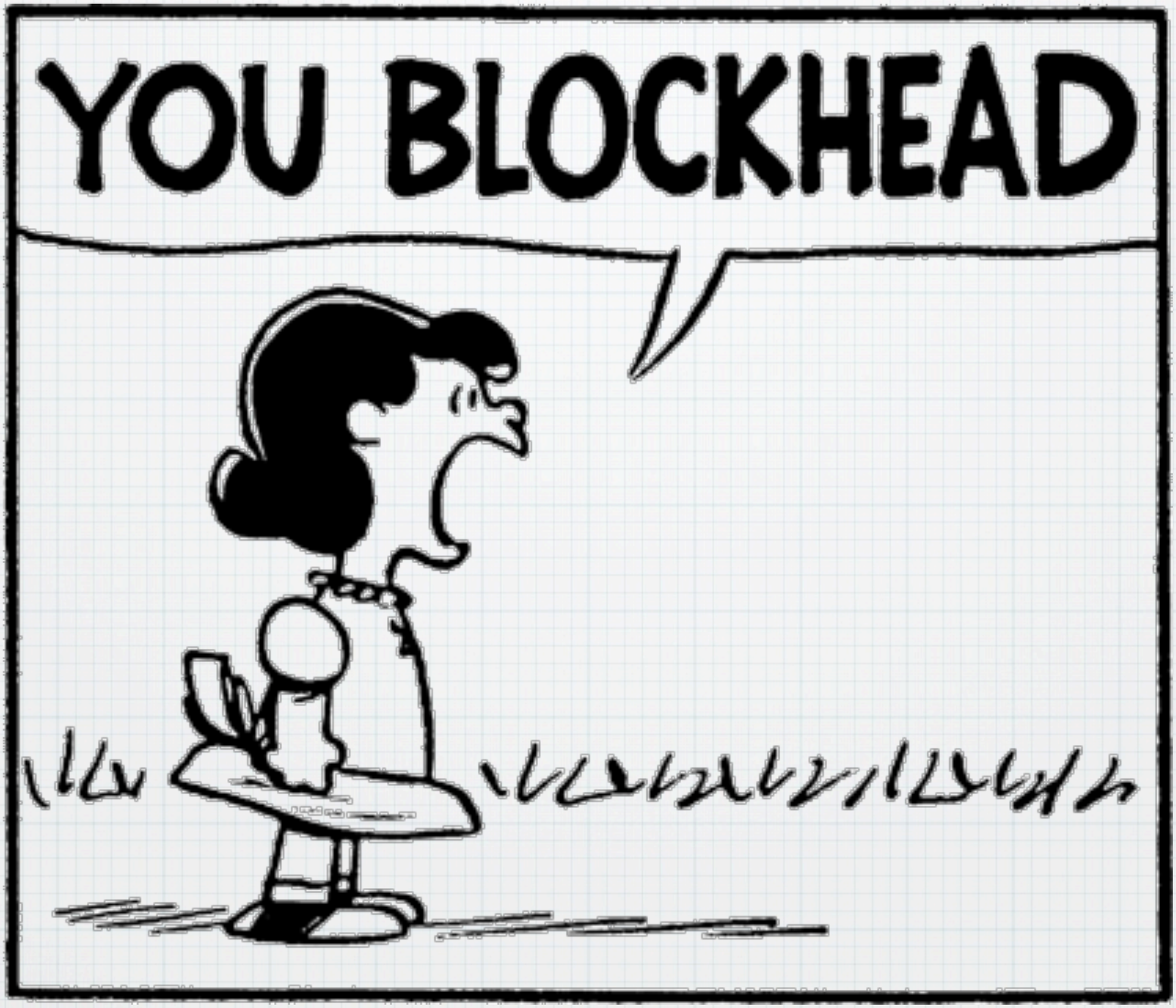
Become a Blockhead

Jack Nutting
@jacknutting
jacknutting@mac.com
http://nuthole.com

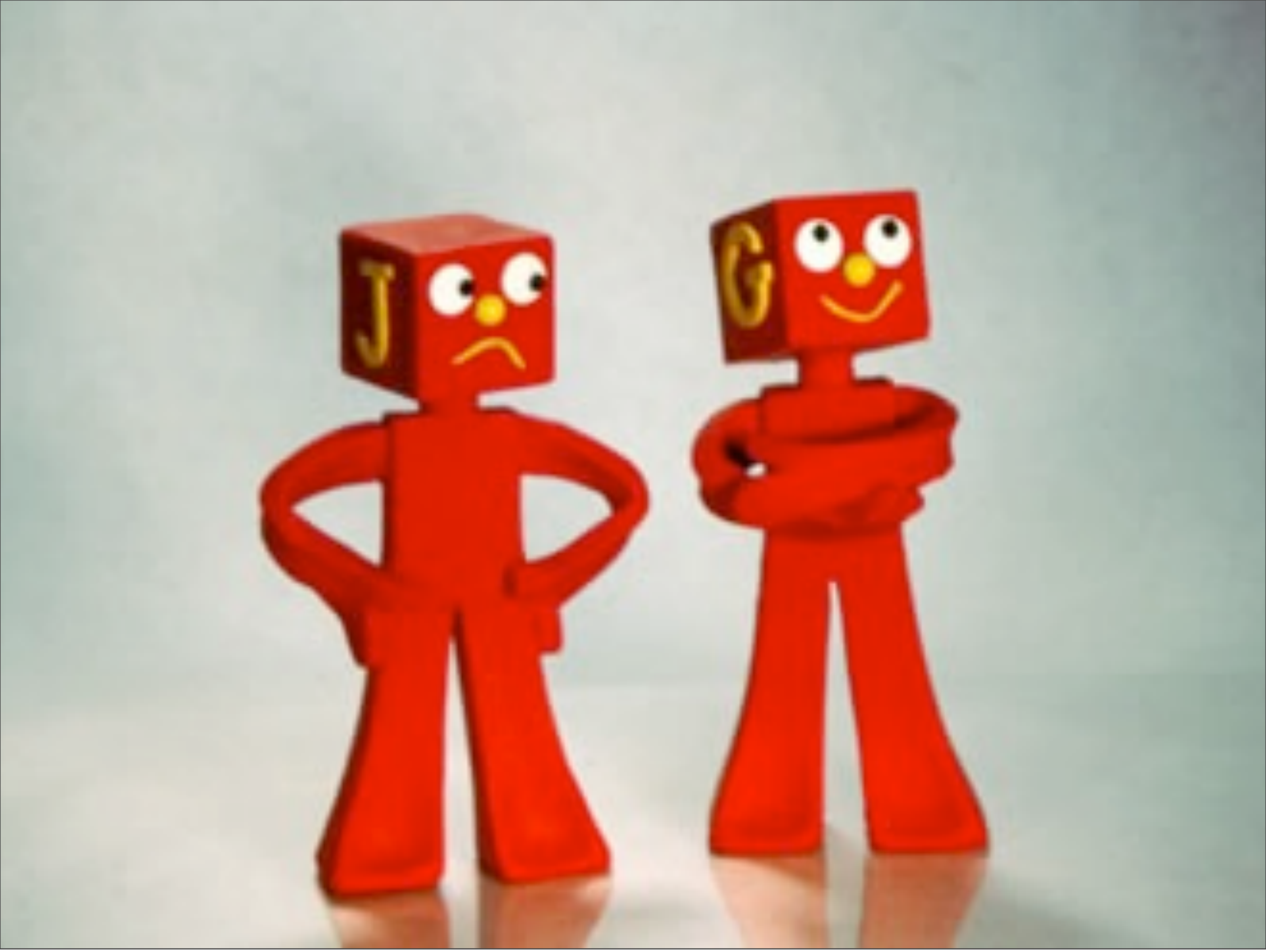
Oredev 2011

Oredev puzzle code: 50709

Blockhead?
What am I talking about?
Wikipedia provides many definitions of “blockhead”...



an idiot



These bad guys from Gumby

Sideshow performer
Or, perhaps scariest of all...

a New Kids on the Block fan

But no. I’m talking about someone who can make sense of...

```
[things each:^(id obj) {
    NSLog(@"Hello %@", obj);
}
```

...this. or...

```
[UIView
 animateWithDuration:1.0
 animations:^{
      view.center = point;
 } completion:^(BOOL finished) {
      button.enabled = YES;
 }];
```

...this. Or even...

```
[request runWithCompletionBlock:^{
    [parser handle:request.data
          yielding:^(JNNode *node) {
              dispatch_async(mainQueue, ^{
                  [node redisplay];
              });
          }];
}];
```

...this. We'll look at this more, but one of the things to look out for is the caret (wedge, hat) symbol...

```
[request runWithCompletionBlock:^{
    [parser handle:request.data
          yielding:^(JNNode *node) {
              dispatch_async(mainQueue, ^{
                  [node redisplay];
              });
          }];
}];
```

... This example has 3, the previous had 2, the earlier had 1. Also note the fun frowny faces...

```
[request runWithCompletionBlock:^{
    [parser handle:request.data
          yielding:^(JNNode *node) {
              dispatch_async(mainQueue, ^{
                  [node redisplay];
              });
          }];
 }];
```

... that appear now and then.
Now, a bit of context...

- Block
- Closure
- Lambda
- Anonymous Function

These are all basically the same thing. Concept exists in scheme and lisp since the 1970s. Most modern languages have them. Different languages have variations, Java and C++ not at all (but soon).
Some common characteristics...

Inline Declaration

```
- (void)touchThemAll:(id)things {
    [things each:^(id obj) {
        NSLog(@"Touching %@", obj);
    }];
}
```

Blocks are typically created inside a method or function.
Can be assigned to a variable, or passed directly to another method or function, as shown here.
Blocks are Objective-C objects, can be retained, released, copied. normally objects are created on heap, but these are created on stack, which has consequences we'll see later.

Variable Capturing

```
int count = 5;
[obj accept:^{
    NSLog(@"This many %d", count);
}];
```

When this code is executed, a new block is created, and the value of the count variable is copied into a new local variable with the same name, inside the block.

Anatomy of a Block

Similarity to C function pointers

```
typedef BOOL (*TruthFunction)(int a, int b);

typedef BOOL (^TruthBlock)(int a, int b);
```

Return type | Type name | Parameters

```
TruthBlock block;

block = ^BOOL(int a, int b) {
    return (a == b);
};

BOOL truth = block(23, 47);
```

It's best to declare a new C type to represent either a function pointer or a block, because it makes variable and parameter declaration simpler. See caret, return type, name of new declared type, parameters, declare variable, assign to variable, execute block stored in variable.
This is a contrived example, you wouldn't normally create a block just to call it immediately.

A better example

```
NSArray *strings = [NSArray arrayWithObjects:
                    @"Hi", @"there,", @"how",
                    @"are", @"you",@"doing?",
                    nil];


[strings enumerateObjectsUsingBlock:
     ^(id obj, NSUInteger idx, BOOL *stop) {
     NSLog(@"%d: %@", idx, obj);
}
```

```
output:  0: Hi
         1: there
         (etc)
```

A bit more realistic. The most common block usage is inline creation, passing them off to another method. enumerateObjectsUsingBlock is a method on NSArray that does just what it says. The block is called once for each element in the array. Explain each parameter

```
lengths = strings.collect { |s| s.length }

bigWords = strings.select { |s| s.length > 3 }
```

What if we want to do more than just enumerate? ruby has nice methods on its collection objects for generating one array from another. collect (map) builds new array from returned values, select builds new array containing just values where block returned true.
NSArray doesn’t have these. Doesn’t have any block-based methods that return an array at all, but we can add them!

Collect

```
lengths = strings.collect { |s| s.length }
```

```
NSArray *lengths = [strings collect:
    ^id(id obj, NSUInteger idx, BOOL *stop) {
    return [NSNumber numberWithUnsignedLong:
                [obj length]];
}];
```

assuming an array of strings, ruby lets us collect the lengths of all the strings into a new array.
We'd like to have a similar Objective-C API.
A bit more verbose than ruby, but it's about the best we can do. To allow greatest flexibility, we support the same parameters that NSArray's other enumeration methods do.

Collect

```
- (NSArray *)collect:
    (id (^)(id obj, NSUInteger idx, BOOL *stop))filter {
    NSMutableArray *collected =
        [NSMutableArray arrayWithCapacity:self.count];
    [self enumerateObjectsUsingBlock:
     ^(id obj, NSUInteger idx, BOOL *stop) {
        [collected addObject:filter(obj, idx, stop)];
    }];
    return collected;
}
```

```
NSArray *lengths = [strings collect:
    ^id(id obj, NSUInteger idx, BOOL *stop) {
    return [NSNumber numberWithUnsignedLong:
            [obj length]];
}];
```

The implementation. Mention caveat about method names in categories on classes you don't own. Once again for context, see how this is used.

Select

```ruby
bigWords = strings.select { |s| s.length > 3 }
```

```objc
NSArray *bigWords = [strings select:
    ^BOOL(id obj, NSUInteger idx, BOOL *stop) {
    return [obj length] > 3;
}];
```

Once again, the ruby syntax.
And about the closest we can come in Objective-C.

Select

```
- (NSArray *)select:
    (BOOL (^)(id obj, NSUInteger idx,
     BOOL *stop))predicate {
    NSIndexSet *indexes =
        [self indexesOfObjectsPassingTest:predicate];
    return [self objectsAtIndexes:indexes];
}
```

```
NSArray *bigWords = [strings select:
    ^BOOL(id obj, NSUInteger idx, BOOL *stop) {
    return [obj length] > 3;
}];
```

again, some context

Build around reliance on delegates

But blocks can be used for much more than just enumeration. You can rethink usage patterns, giving you tighter code that's easier to read.

Delegation is great, but sometimes there's too much. If there's only a few delegate methods required, use a block instead.

NSURLConnection

```
NSURLRequest *req = [NSURLRequest
                     requestWithURL:
                     [NSURL URLWithString:
                      @"http://apple.com"]];

NSURLConnection *conn = [NSURLConnection
                         connectionWithRequest:req
                         delegate:self];
[conn start];
```

NSURLConnection is a prime example. Starting up a requestion is pretty simple. Create a request, then pass it in to a new connection and start it. However...

NSURLConnection

```
- (void)connection:(NSURLConnection *)connection
didReceiveResponse:(NSURLResponse *)response;

- (void)connection:(NSURLConnection *)connection
didReceiveData:(NSData *)data;

- (void)connectionDidFinishLoading:(NSURLConnection
*)connection;

- (void)connection:(NSURLConnection *)connection
didFailWithError:(NSError *)error;
```

...handling the connection via your delegate methods is pretty messy. Must implement at least these methods.

NSURLConnection

```
+ (void)sendAsynchronousRequest:(NSURLRequest *)request
    queue:(NSOperationQueue*) queue
    completionHandler:(void (^)(NSURLResponse*,
                                NSData*, NSError*)) handler;
```

But apple now provides a better way. This method runs in background, provides response, data, and error (if there is one) to handler block. block is called on queue parameter.

NSURLConnection

```
[NSURLConnection sendAsynchronousRequest:req
     queue:[NSOperationQueue mainQueue]
     completionHandler:
 ^(NSURLResponse *response, NSData *data,
    NSError *error) {
      if (data) {
          // success
      } else {
          // examine 'error' and proceed
      }
}];
```

Suddenly all so much simpler. No delegate, no connection to hang onto. Visible chain of cause and effect.

Core Animation

```
[UIView beginAnimations:nil context:NULL];
[UIView setAnimationDuration:0.25];
[UIView setAnimationDelegate:self];
[UIView setAnimationDidStopSelector:
 @selector(animationDidStop:finished:context:)];

self.redView.center = targetPoint;

  [UIView commitAnimations];

- (void)animationDidStop:(NSString *)animationID
                finished:(NSNumber *)finished
                 context:(void *)context {
    // do something else
}
```

Core Animation provides automatic smooth movements by bracketing geometry changes in a certain context. The old way had specific methods to start and end the animation block. If you wanted to be able to execute code when the move was done, you had to implement a delegate method which looked like this.

Core Animation

```
[UIView animateWithDuration:0.25
                 animations:^{
                     self.redView.center = targetPoint;
                 } completion:^(BOOL finished) {
                     // do something else
                 }];
```

Now it’s become a lot easier.

In Your Own Classes

```
@interface MyAsynchronousClass : NSObject
@property (copy) void (^completionBlock)(void);
- (void)doSomethingWithCompletionBlock:
    (void (^)(void))block;
@end
```

A simple example. Declare storage for the block. Must be copy, since blocks are typically created on the stack.

In Your Own Classes

```
@implementation MyAsynchronousClass
@synthesize completionBlock;
- (void)doSomethingWithCompletionBlock:
    (void (^)(void))block {
  self.completionBlock = block;
  [self doPrivateSomething];
}
- (void)doPrivateSomething {
    // start off whatever you're doing in a
    // background thread. At some point when
    // it's all done:
    self.completionBlock();
}
@end
```

In Your Own Classes

```
MyAsynchronousClass *async; // assume this exists

[async doSomethingWithCompletionBlock:^{
    // async action is done, do something interesting
}
```

This is how you use it.

A warning about retain cycles

```
MyAsynchronousClass *async; // this exists as ivar

[async doSomethingWithCompletionBlock:^{
    [self updateDisplay];
}];
```

Assume that async exists, and we “own” it, having a instance variable or property pointing at it.
Each variable used in a block is copied (C values) or retained (objects). This can get us stuck in a retain loop.

The "old" solution, pre-ARC

```
MyAsynchronousClass *async; // this exists as ivar

__block __typeof(self) myself = self;
[async doSomethingWithCompletionBlock:^{
    [myself updateDisplay];
}];
```

Explain __block storage qualifier. Doesn't copy values, provides access to original. In case of objects, provides original pointer, non-retained.
This works for non-ARC code.

The new ARC-friendly solution

```
MyAsynchronousClass *async; // this exists as ivar

__weak __typeof(self) myself = self;
[async doSomethingWithCompletionBlock:^{
    [myself updateDisplay];
}];
```

Change __block to __weak, to force compiler to not retain ‘myself’. This works with ARC. But will it work with ARC and 4.x, is the question?

Nested blocks

```
[request runWithCompletionBlock:^{
    [parser handle:request.data
           yielding:^(JNNode *node) {
               dispatch_async(mainQueue, ^{
                   [node redisplay];
               });
           }];
}];
```

This is a pared-down example from an actual production application. Lots of detail removed, but the core idea is here. Without blocks, this would have required multiple sets of delegates.

Become a Blockhead

Jack Nutting
@jacknutting
jacknutting@mac.com
http://nuthole.com

Oredev 2011

Oredev puzzle code: 50709

Putting up slides today.