Objective-C I

C

- You should understand *everything* in c_example.c.
- Highly recommended that you are in or have taken CS 314 and/or CS 105 C++.

Objective-C

- Superset of C. Extends C by adding syntax for classes and methods.
- Single Inheritance, like Java

Objective-C

- Dynamic runtime. All objects allocated on heap.
- Message (method) dispatch and introspection occur at runtime.

Files

.h	Contains class, function, and constant declarations
.m	Source file, can contain C and Objective-C
.mm	Can contain C++, as well as Objective-C and C

Classes

- Made up of two parts: *interface* and *implementation*

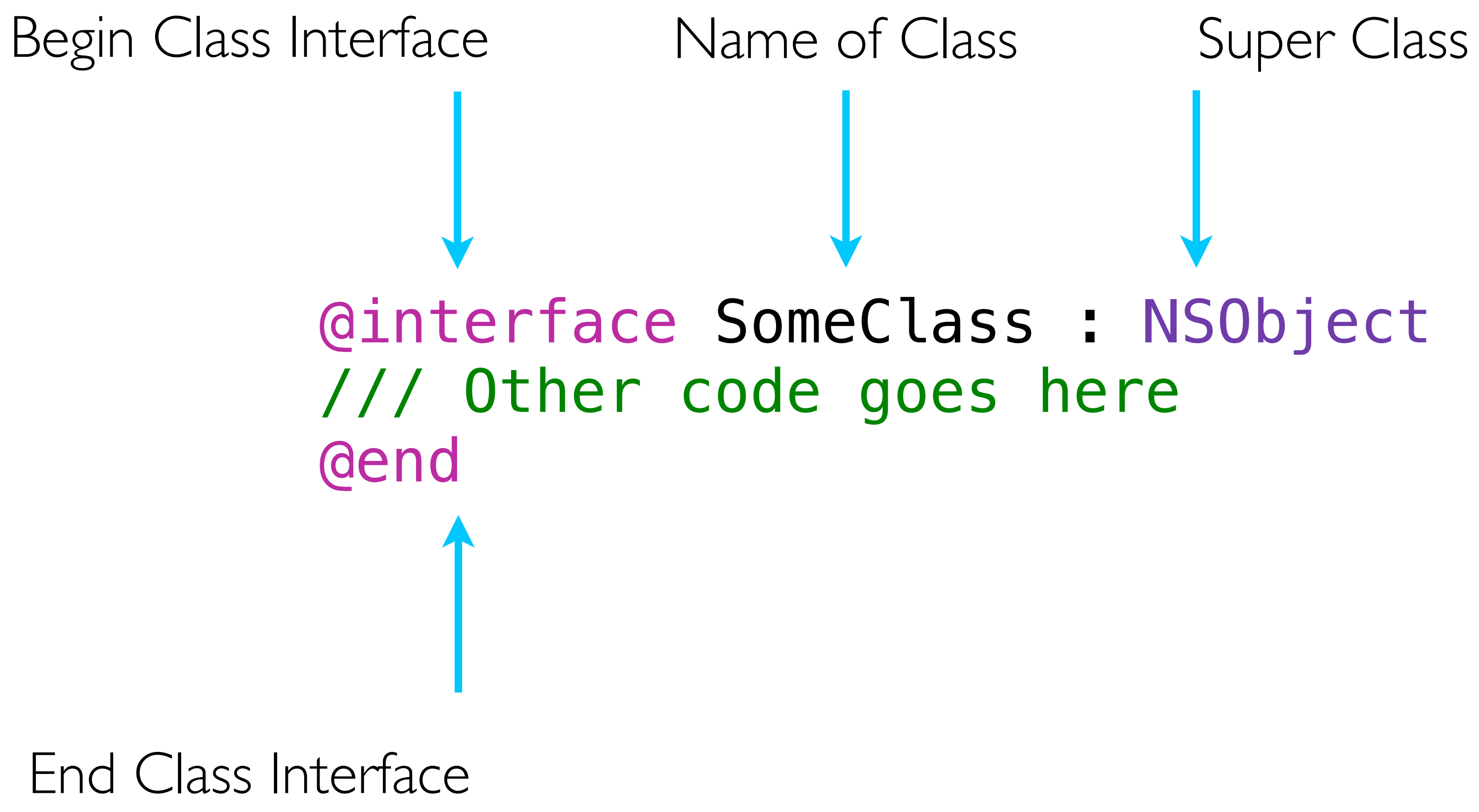
Interface

- Contains the class declaration, defines instance variables and methods associated with class
- Usually in .h file but doesn’t have to be (more on this later)

Implementation

- Contains the actual code for the methods of the class
- Usually in .m file. DO NOT put an implementation anywhere else unless you know what you're doing!

Class Declaration



```
@interface SomeClass : NSObject
/// Other code goes here
@end
```

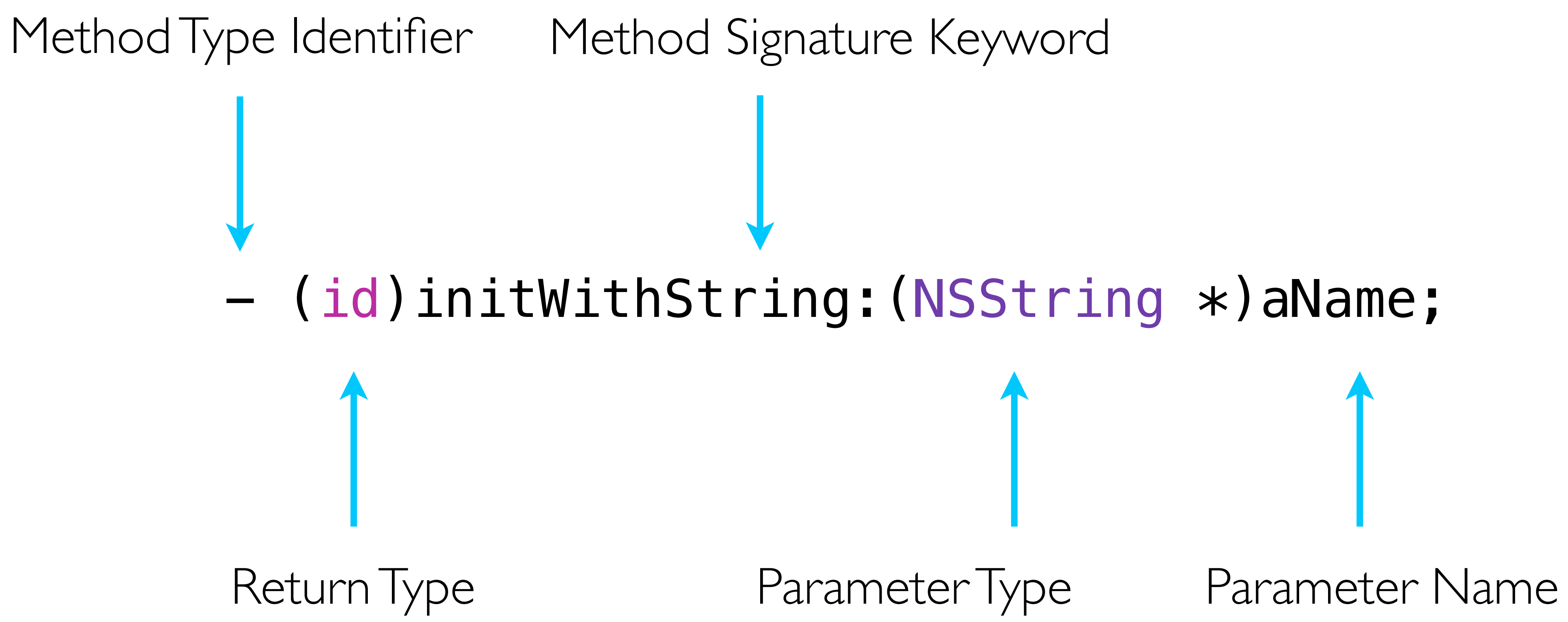
Class Declaration

```
@interface SomeClass : NSObject
{
    /**
      By default, instance variables are
      protected
    */
    int anInt;  /// Instance variable (ivar)
}

@end
```

Class Declaration

```
@interface SomeClass : NSObject
{
    int anInt;
}

/**
  Methods declared in class interface are always public.
*/

/// Instance Method
- (id)initWithString:(NSString *)aName;

/// Class Method
+ (SomeClass *)createSomeClassWithString:(NSString *)aName;

@end
```

Method Declarations



```
- (id)initWithString:(NSString *)aName;
```

Method Type Identifier

Method Type Identifier

```
            - (id)initWithString:(NSString *)aName;

+ (SomeClass *)createSomeClassWithString:(NSString *)aName;
```

Instance method declarations begin with -

Class method declarations begin with +

Method Signatures

Method signatures can be very simple:

```
- (id)initWithString:(NSString *)aName;
```

Or very complex:

```
- (id)initWithString:(NSString *)aName
  count:(int)anInt data:(id)someData;
```

Parameters are embedded in the method name!

Compare to Java

```
/** Java */
public void distanceFromObject(SomeObject object, int time)
{
/// Code
}

/** Objective-C */
- (void)distanceFromObject:(SomeObject *)object atTime:(int)time
{
/// Code
}
```

Method's actual name is a concatenation of all the signature keywords:

```
distanceFromObject:atTime:
```

Pulse Check

- What are the two method types?
- What's the difference when declaring them?
- Class implementations should be in files with what extension(s)?
- By default, are instance variables public, private, or protected?
- I want to declare a constant. Where should I put it?

Implementation

```
#import “SomeClass.h”

@implementation SomeClass

- (id)initWithString:(NSString *)aName
{
    /// Code goes here
}

+ (SomeClass *)createSomeClassWithString:(NSString *)aName
{
    /// Code goes here
}

@end
```

#import

- Objective-C's `#import` is similar to C's `#include` except it guarantees files are only included once.
- Always prefer `#import`

Messaging

- When calling a method, you do so by *messaging* an object.
- A message consists of the method signature along with method parameters.
- All messages dispatched dynamically. Achieves dynamic binding.

Messaging

- When calling an object's method, you do not need to know the object type. The method has to exist.
- This is known as message passing.

Messaging

```
/** Java */
someObject.message();


/** Objective-C */
[someObject message];
```

Messaging

```
/** Java */
someObject.message(argument);


/** Objective-C */
[someObject message:argument];
```

Messaging

```
/** Java */
someObject.message(argOne, argTwo);


/** Objective-C */
[someObject message:argOne withSecondArg:argTwo];
```

Nesting Messages

- Messages can be nested, meaning the result of one message can be used in another message

```
/** Java */
someObject.message(otherObject.result());

/** Objective-C */
[someObject message:[otherObject result]];
```

Passing Messages

- What if you pass a message to a `nil` object?
- First, what the hell is `nil`?

Passing Messages

- `nil` is the Objective-C equivalent of `NULL`
- Nothing happens if you pass a message to a nil object!

Passing Messages

What happens here?

```
- (void)someMethod:(SomeObject *)someObject
{
    someObject = nil;
    NSString *name = [someObject name];
    if(name == nil) {
        NSLog(@“%@”, @“Name is nil”);
    }
}
```

Brownie points if you can guess what NSLog does.

id

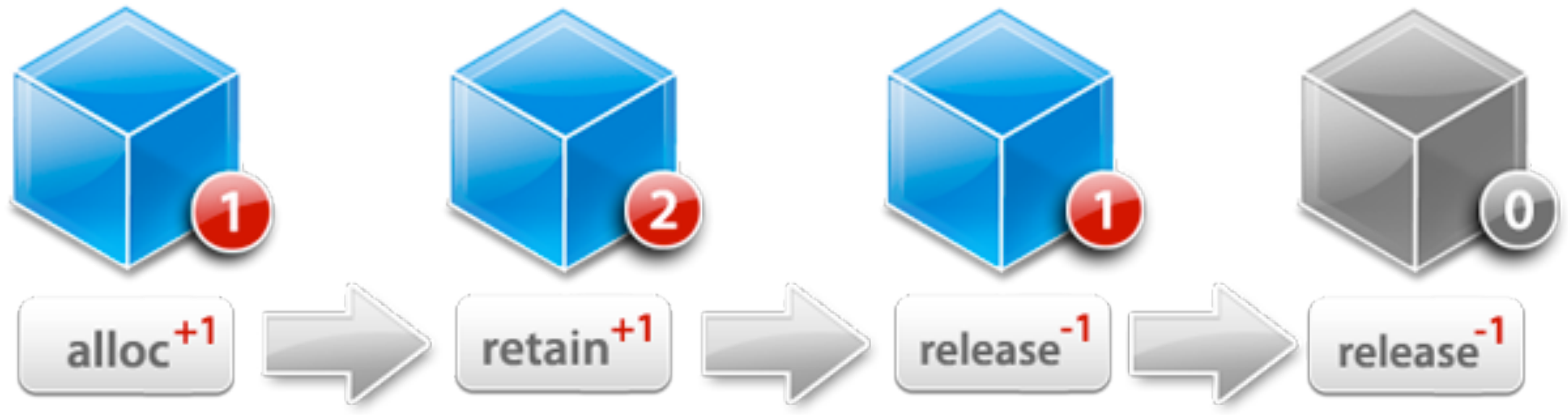
- In Objective-C, `id` is a special type that acts very similarly to `void*`
- `id` can only be used for objects

id Example Usage

```
#import “SomeClass.h”

@implementation SomeClass

- (NSString *)classNameForObject:(id)someObject
{
    return NSStringFromClass([someObject class]);
}

@end
```

- Notice that we did not need to know the type for `someObject`

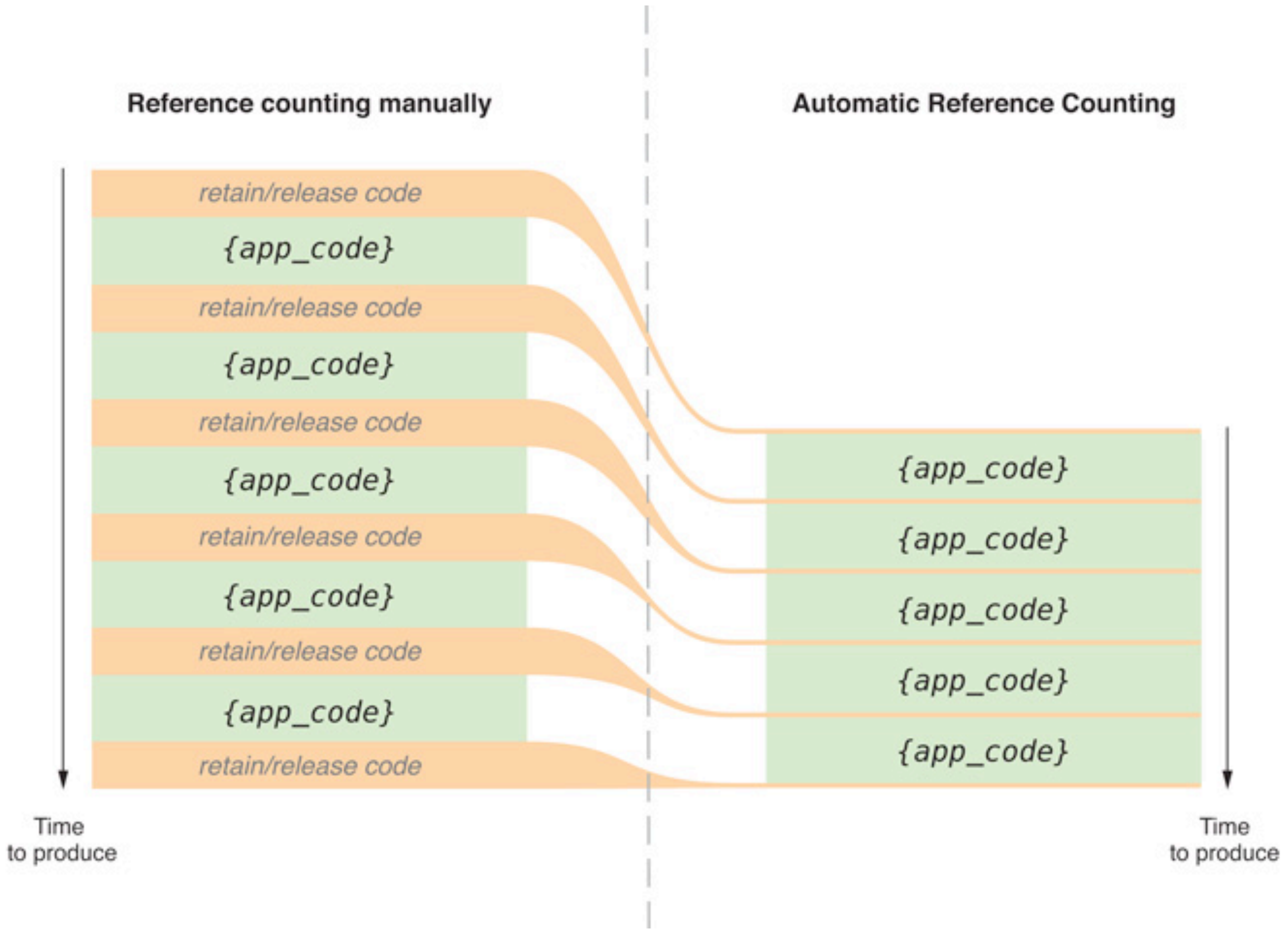
Memory Management



Memory Management

- Nope, just kidding
- Automatic Reference Counting (ARC) was implemented by Apple. This injects code at compile to handle deallocating objects when they are no longer being referenced.
- ARC is a form of garbage collection

ARC


Reference counting manually
Automatic Reference Counting
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
{app_code}
{app_code}
{app_code}
{app_code}
Time to produce
Time to produce

That’s All Folks