Difference between nullable, __nullable and _Nullable in Objective-C

Asked 6 years, 7 months ago Modified 1 year, 7 months ago Viewed 45k times

167

90

With Xcode 6.3 there were new annotations introduced for better expressing the intention of API's in **Objective-C** (and to ensure better Swift support of course). Those annotations were of course `nonnull`, `nullable` and `null_unspecified`.

But with Xcode 7, there is a lot of warnings appearing such as:

> Pointer is missing a nullability type specifier (_Nonnull, _Nullable or _Null_unspecified).

In addition to that, Apple uses another type of nullability specifiers, marking their C code (source):

```
CFArrayRef __nonnull CFArrayCreate(CFAllocatorRef __nullable allocator, const void * __nonnull * __nullable values, CFIndex numValues, const CFArrayCallBacks * __nullable callBacks);
```

So, to sum up, we now have these 3 different nullability annotations:

- `nonnull`, `nullable`, `null_unspecified`
- `_Nonnull`, `_Nullable`, `_Null_unspecified`
- `__nonnull`, `__nullable`, `__null_unspecified`

Even though I know why and where to use which annotation, I'm getting slightly confused by which type of annotations should I use, where and why. This is what I could gather:

- For properties I should use `nonnull`, `nullable`, `null_unspecified`.
- For method parameters I should use `nonnull`, `nullable`, `null_unspecified`.
- For C methods I should use `__nonnull`, `__nullable`, `__null_unspecified`.
- For other cases, such as double pointers I should use `_Nonnull`, `_Nullable`, `_Null_unspecified`.

But I'm still confused as to why we have so many annotations that basically do the same thing.

So my question is:

What is exact difference between those annotations, how to correctly place them and why?

objective-c nullable objective-c-nullability

Share Edit Follow

edited Sep 12, 2017 at 15:18
Cœur
34.4k ● 23 ● 184 ● 249

asked Sep 8, 2015 at 8:29
Legoless
10.7k ● 7 ● 45 ● 67

3 I've read that post, but it does not explain the difference and why we have 3 different types of annotations now and I really want to understand why they went on adding the third type. – Legoless Sep 8, 2015 at 8:34

2 This really does not help @Cy-4AH and you know it. :) – Legoless Sep 8, 2015 at 9:00

@Legoless, are you sure you read it carefully? it explains precisely where and how you should use them, what are the audited scopes, when you can use the other one for better readability, compatibility reasons, etc, etc... you might not know what you really like to ask, but the answer is clearly under the link. it is maybe just me, but I don't feel that any further explanation would be necessary for explaining their purpose, copying and pasting that simple explanation to here as an answer here would be really awkward, I guess. :(– holex Sep 8, 2015 at 9:23

2 It does clear up certain parts, but no, I still don't understand why we don't just have the first annotations. It only explains why they went from __nullable to _Nullable, but not why do we even need _Nullable, if we have nullable. And it also does not explain why Apple still uses __nullable in their own code. – Legoless Sep 8, 2015 at 9:46

4 Answers

Sorted by: Highest score (default)

Help us improve our answers.

Are the answers below sorted in a way that puts the best answer at or near the top?

Take a short survey I'm not interested

173

From the `clang` documentation:

> The nullability (type) qualifiers express whether a value of a given pointer type can be null (the `_Nullable` qualifier), doesn't have a defined meaning for null (the `_Nonnull` qualifier), or for which the purpose of null is unclear (the `_Null_unspecified` qualifier). Because nullability qualifiers are expressed within the type system, they are more general than the `nonnull` and `returns_nonnull` attributes, allowing one to express (for example) a nullable pointer to an array of nonnull pointers. Nullability qualifiers are written to the right of the pointer to which they apply.

, and

> In Objective-C, there is an alternate spelling for the nullability qualifiers that can be used in Objective-C methods and properties using context-sensitive, non-underscored keywords

So for method returns and parameters you can use the the double-underscored versions `__nonnull`/`__nullable`/`__null_unspecified` instead of either the single-underscored ones, or instead of the non-underscored ones. **The difference is that the single and double underscored ones need to be placed after the type definition, while the non-underscored ones need to be placed before the type definition.**

Thus, the following declarations are equivalent and are correct:

```
- (nullable NSNumber *)result
- (NSNumber * __nullable)result
- (NSNumber * _Nullable)result
```

For parameters:

```
- (void)doSomethingWithString:(nullable NSString *)str
- (void)doSomethingWithString:(NSString * _Nullable)str
- (void)doSomethingWithString:(NSString * __nullable)str
```

For properties:

```
@property(nullable) NSNumber *status
@property NSNumber *__nullable status
@property NSNumber * _Nullable status
```

Things however complicate when double pointers or blocks returning something different than void are involved, as the non-underscore ones are not allowed here:

```
- (void)compute:(NSError *  _Nullable * _Nullable)error
- (void)compute:(NSError *  __nullable * _Null_unspecified)error;
// and all other combinations
```

Similar with methods that accept blocks as parameters, please note that the `nonnull`/`nullable` qualifier applies to the block, and not its return type, thus the following are equivalent:

```
- (void)executeWithCompletion:(nullable void (^)())handler
- (void)executeWithCompletion:(void (^ _Nullable)())handler
- (void)executeWithCompletion:(void (^ __nullable)())handler
```

If the block has a return value, then you're forced into one of the underscore versions:

```
- (void)convertObject:(nullable id __nonnull (^)(nullable id obj))handler
- (void)convertObject:(id __nonnull (^ _Nullable)())handler
- (void)convertObject:(id _Nonnull (^ __nullable)())handler
// the method accepts a nullable block that returns a nonnull value
// there are some more combinations here, you get the idea
```

As conclusion, you can use either ones, as long as the compiler can determine the item to assign the qualifier to.

Share Edit Follow

edited Mar 30, 2016 at 20:07

answered Nov 12, 2015 at 21:59
Cristik
28.3k ● 24 ● 83 ● 117

3 It seems that the underscores versions can be used everywhere, so I suppose I will consistently use them, rather than using underscored versions in some places and ones without the underscore in others. Correct? – Kartick Vaddadi Apr 22, 2016 at 12:47

@KartickVaddadi yes, correct, you can consistently use either the single underscored, or the double underscored versions. – Cristik Apr 22, 2016 at 12:51

`_Null_unspecified` in Swift this translates to optional? non-optional or what? – mfaani Jan 30, 2018 at 20:21

1 @Honey _Null_unspecified is imported in Swift as Implicitly Unwrapped Optional – Cristik Jan 30, 2018 at 20:36

1 @Cristik aha. I guess that's its default value...because when I **didn't** specify I was getting Implicitly unwrapped optional... – mfaani Jan 30, 2018 at 20:44

30

From the Swift blog:

> This feature was first released in Xcode 6.3 with the keywords __nullable and __nonnull. Due to potential conflicts with third-party libraries, we've changed them in Xcode 7 to the _Nullable and _Nonnull you see here. However, for compatibility with Xcode 6.3 we've predefined macros __nullable and __nonnull to expand to the new names.

Share Edit Follow

edited May 27, 2016 at 19:07
manroe
1,455 ● 17 ● 30

answered Nov 23, 2015 at 4:14
Ben Thomas
1,443 ● 1 ● 16 ● 25

4 In a nutshell, the single and double underscored versions are identical. – Kartick Vaddadi Apr 22, 2016 at 12:44

4 Also documented in Xcode 7.0 release notes: "The double-underscored nullability qualifiers (__nullable, __nonnull, and __null_unspecified) have been renamed to use a single underscore with a capital letter:_Nullable, _Nonnull, and _Null_unspecified, respectively). The compiler predefines macros mapping from the old double-unspecified names to the new names for source compatibility. (21530726)" – Cosyn Jun 24, 2016 at 3:20

30

From the clang documentation:

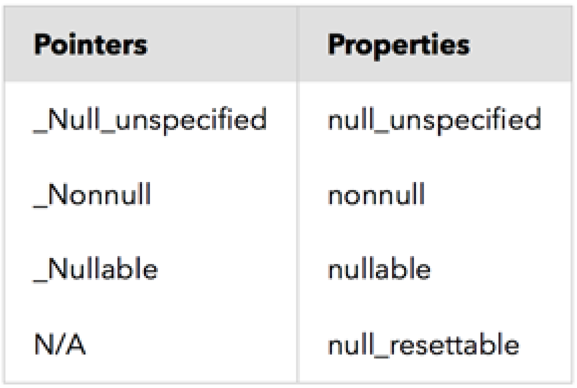
> The nullability (type) qualifiers express whether a value of a given pointer type can be null.

Most of the time you will use `nonnull` and `nullable`.

Below are all the available specifiers. From **this article**:

- `null_unspecified:` **This is the default.** It bridges to a Swift implicitly-unwrapped optional.
- `nonnull`: the value won't be nil. It bridges to a Swift regular reference.
- `nullable`: the value can be nil. It bridges to a Swift optional.
- `null_resettable`: the value can never be nil when read, but you can set it to nil to reset it. Applies to properties only.

The notations above differ whether you use them in the *context* of properties or functions/variables:



Pointers	Properties
_Null_unspecified	null_unspecified
_Nonnull	nonnull
_Nullable	nullable
N/A	null_resettable

The author of the article also provided a nice example:

```
// property style
@property (nonatomic, strong, null_resettable) NSString *name;
```

```
// pointer style
+ (NSArray<NSView *> * _Nullable)interestingObjectsForKey:(NSString *
_Nonnull)key;

// these two are equivalent!
@property (nonatomic, strong, nullable) NSString *identifier1;
@property (nonatomic, strong) NSString * _Nullable identifier2;
```

Share Edit Follow

edited Sep 8, 2020 at 13:30

answered Jan 7, 2016 at 20:20

kgaidis
12.1k ● 4 ● 69 ● 82

13

Very handy is

```
NS_ASSUME_NONNULL_BEGIN
```

and closing with

```
NS_ASSUME_NONNULL_END
```

This will nullify the need for the code level 'nullibis' :-) as it sort of makes sense to assume that **everything** is non-null (or `nonnull` or `_nonnull` or `__nonnull`) unless otherwise noted.

Unfortunately there are exceptions to this as well...

- `typedef`s are not assumed to be `__nonnull` (note, `nonnull` does not seem to work, have to use it's ugly half brother)
- `id *` needs an explicit nullibi but wow the sin-tax (`_Nullable id * _Nonnull` <- guess what that means...)
- `NSError **` is always assumed nullable

So with the exceptions to the exceptions and the inconsistent keywords eliciting the same functionality, perhaps the approach is to use the ugly versions `__nonnull` / `__nullable` / `__null_unspecified` and swap when the complier complains... ? Maybe that is why they exist in the Apple headers?

Interestingly enough, something put it into my code... I abhor underscores in code (old school Apple C++ style guy) so I am absolutely sure I did not type these but they appeared (one example of several):

```
typedef void ( ^ DidReceiveChallengeBlock ) (
NSURLSessionAuthChallengeDisposition disposition,
                                          NSURLCredential * __nullable
credential );
```

And even more interestingly, where it inserted the __nullable is wrong... (eek@!)

I really wish I could just use the non-underscore version but apparently that does not fly with the compiler as this is flagged as an error:

```
typedef void ( ^ DidReceiveChallengeBlock ) (
NSURLSessionAuthChallengeDisposition disposition,
                                             NSURLCredential * nonnull  credential
);
```

Share Edit Follow

edited Jul 20, 2017 at 5:31
Hans Knöchel
11.3k ● 8 ● 27 ● 48

answered Nov 18, 2015 at 14:03
Cerniuk
12.9k ● 2 ● 27 ● 27

2 Non-underscore can only be used directly after an opening parenthesis, i.e. (nonnull ... Just to make life more interesting, I'm sure. – Elise van Looij Feb 7, 2017 at 22:08

How do I make an id<> nonnull? I feel like this answer contains a lot of knowledge but it lacks clarity. – fizzybear Nov 8, 2019 at 2:05

1 @fizzybear admittedly I typically take the opposite approach. Pointers are my friend and I have not had "null" / "nil" pointer problems since the early 90's. I wish I could make the whole nullable/nilable thing just go away. But to the point, the real answer is in the first 6 lines of the answer reply. But about your question: not something I would do (no criticism) so I just don't know. – Cerniuk Nov 8, 2019 at 20:30