The following appear to be minor issues in *The Swift Programming Language (Swift 5.7)* document. The edition reviewed is the ePub download from swift.org, not the online version. These issues are identified for possible resolution in the release of this document for a subsequent version of Swift.

* As a general comment, it appears that the textual updates for actors were scattershot as opposed to thoroughly integrated.
  + It might be worthwhile to explicitly summarize how actors are similar to and different from classes, as is done for classes, enumerations and structures.
  + It might be worthwhile to review the text to where class, enumeration and structure are referenced together. In some, but perhaps not all, of these cases, actor should be added.
  + It is unclear whether an actor instance utilizes ARC.
    - It is unclear whether references to an actor can be **unowned** or **weak**.
  + It is unclear whether an actor can have a deinitializer.
    - This is currently allowed in Xcode although according to the Deinitialization section it shouldn’t be.
      * *Given the nature and purpose of actors, it could be that Xcode yields the desired behavior, provided that the deinitializer is ever invoked. Alternatively, it could be that the change occurred too late to be updated in the text or was merely overlooked.*
  + It is unclear whether actor is a first-class type.
  + It is unclear what access level rules apply to actors as they do vary between classes, enumerations and structures.
  + The final paragraph in the Actor Declaration section probably intended to state actor in lieu of structure type or that paragraph was inadvertently duplicated due to a cut / paste error.
  + **distributed** should be a keyword, a declaration modifier, incorporated into the grammar and warrant discussion in both the [Language Guide] Concurrency and [Language Reference] Declaration Modifiers sections
    - In the WWDC 2022 video *What’s New in Swift*, there is a brief section titled Distributed actors at about time 16:40
      * In the example code provided, both **actor** and **func** are modified by **distributed**
      * The discussion refers to this as a new language feature as well as a core language primitive
  + It is unclear whether **actor** should be one of the import-kind options in the Grammar of an Import Declaration
  + It is unclear whether await must be used for every separately referenced var accessible outside of the actor.
    - For example, is the await keyword required to be given ahead of an assignment that sets that var? Would such an await also handle any reference of that var or other vars from the same actor in the expression being evaluated for assignment? Can a single await cover vars from multiple actors? Can a pair of actors create a deadlock? If not, what runtime behavior should be expected to occur?
  + It is unclear whether any actor’s var could be manipulated via an escaped closure defined in the actor, but invoked outside of the actor.
  + It is unclear when actor A has logic that effects actor’s B mutable state, whether an await on actor A will implicitly await on actor B.
    - If not, this could yield a deadlock.
    - It is unclear whether the compiler detects cycles amongst actors and generates errors to prevent them, i.e., actor A references actor B while actor B references actor A.
  + It is unclear whether a class-only protocol can be adopted by an actor.
    - Only structures and enumerations are explicitly prohibited. As both are value types and an actor is a reference type, it is unclear whether this restriction is intended for just value types or any non-class type, particularly given the note at the end of the Class-Only Protocols section.
  + It is unclear whether an actor can be generic.
  + It is unclear whether an extension may be applied to an actor.
  + It is unclear whether an actor can have a subscript.
  + It is unclear whether an actor may have a property wrapper defined upon it.
* As a general comment, there are a few critical items given in the Language Reference section that should at least be referenced in passing or with a note in the Language Guide.
  + It might be worthwhile to add a section to the Language Guide for regular expressions and briefly mention or touch on them in the A Swift Tour section
  + It might be worthwhile to add a section or note to the Language Guide for compiler control statements.
  + It might be worthwhile to add a section to the Language Guide with respect to interfacing with Objective-C code and the language features provided for that and then link to an appropriate document for more details.
  + It might be worthwhile to add a note to the Language Guide for metatype types.
  + It might be worthwhile to add a section or note to the Language Guide for key-path expressions.
  + It might be worthwhile to add a section or note to the Language Guide for what is discussed in the Methods with Special Names section
* Potential errors
  + On page 540, in the overview of Attributes, the text indicates that there are two types, declaration and type. However, a section for Switch Case Attributes is given on page 564.
    - *Page numbers will vary depending on the window size open. These are intended to provide relative locations, i.e., earlier or later in the text.*
  + A **do**-**catch** statement may be labeled and contain a **break** statement that references the label, but is not so identified in the Label Statement subsection of the Language Guide section. It is, however, identified in the Label Statement subsection of the Language Reference section.
    - *Note: XCode incorrectly states “Labels are only valid on loops, if and switch statements”, but does allow them on* ***do****-****catch*** *statements*.
  + Indentation of the **do**-**catch** statement on page 488 in the Do Statement section is inconsistent with respect to the last three lines
    - Same issue appears in the **do**-**catch** statement illustration on page 287 in the Handling Errors Using Do-Catch section
  + The section title should be *Chaining on Functions with Optional Return Values*, not *Chaining on Methods with Optional Return Values* as the leftmost portion of the optional chaining expression could be a global function.
    - Methods are functions, but not necessarily the reverse.
  + The final paragraph in the Precedence and Associativity section implies that the only precedence groups available are those discussed in the Standard Library article Operator Declarations, ignoring the Precedence Group Declaration given in the Language Reference that allows additional precedence groups to be defined.
  + In the Keywords and Punctuation section
    - **private** and **precedencegroup** are reversed in the Keywords used in declarations list.
      * **actor** and **nonisolated** should also be in the same list
    - **throw** and **switch** are reversed in the Keywords used in statements list.
    - **await** and **async** should be added to the Keywords used in expressions and types list.
    - **#elseif** and **#else** are reversed in the Keywords that begin with a number sign (#) list.
      * **#file** should precede **#fileID** in the same list.
      * **#unavailable** should also be in the same list given the discussion on page 123 in the Checking API Availability section and its inclusion in the grammar on page 589 in the Grammar of an Availability Condition section
    - **assignment**, **higherthan**, **lowerthan**, **safe** and **unsafe** should be added to the Keywords reserved in particular contexts list.
    - Either the following should be added to a new list, perhaps Keywords in compilation conditions or they should be added to the Keywords reserved in particular contexts list: **arm**, **arm64**, **arch**, **canImport**, **compiler**, **i386**, **iOS**, **iOSApplicationExtension**, **Linux**, **macCatalyst**, **macCatalystApplicationExtension**, **macOS**, **macOSApplicationExtension**, **os**, **simulator**, **swift**, **targetEnvironment**, **tvOS**, **tvOSApplicationExtension**, **watchOS**, **watchOSApplicationExtension**, **Windows** and **x86\_64**.
  + **precedence** should be removed from the Keywords reserved in particular contexts list in the Keywords and Punctuation section.
  + Is the final sentence in the third paragraph of the Protocol Composition Type section correct? Shouldn’t it be “… declaring a new **class** …” rather than “… declaring a new **protocol** …”?
  + The Type Alias Declaration section discussion in the Language Reference seems to omit the use for a protocol composition as well as for tuples. The grammar doesn’t seem to cover either as well.
  + In the Declaration Modifiers section, reword the first sentence in the descriptions of **unowned** and **unowned(unsafe)**.
    - Constant is referred to in the first half of the sentence, but not the second. The sentence should balance.
  + The keyword **any** is missing from the grammar, along with a description of its semantics and any applicable guidance.
* Clarifications requested
  + It is unclear whether the Operator Methods discussion on pages 141 and 142 applies to unary operators or is restricted to non-assignment binary operators.
  + It is unclear what the functional difference between the bitwise shift right ( >> ) and bitwise shift right ignoring overflow ( &>> ) operators is.
  + In the Swift Standard Library documentation, pointwise operators are included in the discussion of operator declarations with respect to precedence and associativity, but they don’t appear in book. If they are part of the core language, then they should be discussed in the book.
    - The pointwise unary operator seems to be inconsistent with other pointwise operators as the pointwise AND, OR and XOR operators are referred to as bitwise. Shouldn’t the unary operator also be so identified and shouldn’t it be **.~**? Or would that be required as an additional operator to complete the pointwise bitwise set? Further, given there are pointwise comparison operators other than equality and inequality, why aren’t there the nominal arithmetic operators (addition, subtraction, multiplication, division and remainder)?
  + It is unclear whether the less than ( < ) operator may be overloaded.
  + It is unclear whether the **inout** modifier may be used in overloading operator mdethods or is restricted to custom operators.
  + It is unclear whether the **infix** modifier may be used in the type method for defining an overloaded or custom operator
  + It is unclear whether an enumeration is considered an ordered or unordered collection when it adopts the **CaseIterable** protocol.
  + It might be worthwhile to explicitly state that an enumeration may have raw values or associated values, but not both.
  + It might be worthwhile to explicitly state that a default value may be provided for an enumeration case associated value, allowing that associated value not to be required when an instance of that enumeration case is created.
  + It is unclear whether a constant may be overridden by a variable and vice versa.
  + Given that both the name and type of an overriding property must be specified, it is unclear what the result of a corresponding name with a different type. Further when the type of the overridden type is a class, can the type of the overriding type be given as a subclass of that?
  + A note indicates that when a property setter is overridden, a getter must also be provided. It is unclear whether the reverse is also the case when the overridden property has a setter.
  + It might be worthwhile in the Preventing Overrides section to note that type methods and properties defined as **static** also cannot be overridden.
  + It is unclear whether a protocol inheritance is required to be acyclic.
    - Cycle protocol inheritance would mean one protocol (A) can inherit another protocol (B) that directly or indirectly inherits the original protocol (A).
    - If this is allowed, it causes any protocol that inherits from A to also inherit all protocols in the inheritance cycle from A back to A, essentially making them all the same.
  + It is unclear what ***protocol extend*** refers to in the following from the final paragraph of the Protocol Extensions section: Protocol extensions can add implementations to conforming types but can’t make a ***protocol extend*** or inherit from another protocol. Is this simply stating that all requirements must be given in the protocol’s declaration? Or is this redundant with the phrase that follows, i.e., that an extension to a protocol can’t add inheritance to another protocol?
  + It might be worthwhile in the Protocol section with a note or subsection, indicate that protocols don’t follow the nominal generic type parameter syntax employed elsewhere in the language, but rather use associated types and a link to the Associated Types section.
  + It might be worthwhile in the Type Constraint Syntax section to add a note or paragraph that points out that using this syntax only a single constraint may be given for each type parameters, but when multiple constraints are required and this syntax must be used, i.e., for a generic function, a type alias may be used with protocol composition to provide the composite constraint.
  + It might be worthwhile to explicitly summarize rules regarding custom operators
    - Whether a compound operator may be defined for a custom operator
      * If so, are all custom operators whose final character is an equal sign compound operators
      * Which associativities are allowed for compound operator precedence groups
      * Whether the method for a custom compound operator may have a return type other than Void or Void?
    - Whether the method for a custom operator may return an optional
    - Whether the second parameter of an infix operator may be in-out
    - Whether the first parameter for a custom operator other than compound may be in-out
    - Whether the method for a custom operator may be async
      * If so, must that be the final operator evaluated in an expression
  + It might be worthwhile to identify any limitations on override methods (such as some that may be listed above)
  + It might be worthwhile to note that a cast operator is merely represented by an init of the casting type whose sole parameter is that of the casted type
  + It might be worthwhile to reword the text in the Generics subsection of Access Control.
    - The terms generic type and generic function in that sentence appear twice with slightly different meanings, the first instance being the entity bound to parameter types and the second instance not being.
  + It is unclear what the default access level for a type alias is.
    - Given the rules stated, and the note in particular, this is important.
  + It is unclear whether a value is missing from the choices for the **os()** platform condition or whether ipadOS is considered to be included in iOS.
  + It is unclear whether a value is missing from the choices for the **targetEnvironment()** platform condition.
  + It might be worthwhile indicating whether a compilation directive must be on its own line(s).
  + It might be worthwhile to add a note, brief section or a paragraph to an existing section that discusses uninhabited types.
    - I didn’t encounter the term in the existing document, but did in some articles and, if the information in those articles is correct, consider it to be a core language feature.
* Consistency
  + Identify **[** **]** as either square braces or square brackets throughout the text, not both
  + Identify **{** **}** as either braces or curly braces throughout the text, not both
  + Refer to it either as a **do** statement or a **do**-**catch** statement, but not both.
  + Refer to it either as a **for**-**in** loop, a **for** statement or a **for**-**in** statement, but not all three.
  + Refer to it either as a branch statement or a conditional statement, but not both.
  + In the Statements section, there is a definition of simple statements being declarations and expressions. In the subsequent Top-Level Code section, it indicates that a Swift program consists of zero or more statements**, declarations and expressions**. The portion marked in red is redundant as declarations and expressions are simple statements.
  + Sometimes a referenced attribute is given a leading commercial at ( **@** ) in the text, but mostly not.
* It might be worthwhile to add a Glossary section where each term defined in the document is given, along with its definition and hyperlinked back to that definition in context.
  + The search capability will locate every instance of a search string, but for terms that are short or are the same as a keyword, this is often not useful to locate the meaning.
  + This only need apply to terms currently explicitly defined in the text. Such terms are already given in italics.
  + Fundamentally, this is no different than what is already done for grammars in the Summary of the Grammar section.
* It might be worthwhile to add a Best Practices or Coding Guidelines sections that captures all of the conventions and suggestions already given in the existing text.
  + These do not necessarily have a common term such as should or shall that may be readily searched for
* Be aware that depending on the window width employed, multicharacter operators may be displayed partially on two lines, e.g., the – of -> on one line and the > on the next. If someone used this as a model for coding, the compiler would generate an error.
  + I don’t know if some formatting may be used to prevent this
  + I don’t know if this only applies to the ePub version or may also occur with the online document viewed with a browser