Omit-needless-words

The name translation rules for Objective-C methods and properties are described in terms of a process called "omit needless words". This is a complicated series of heuristics tuned to the Coding Guidelines for Cocoa, particularly as applied to Apple's SDKs. The full specification for this is, unfortunately, the implementation (in lib/Basic/StringExtras.cpp), because any changes can affect source compatibility.

If you're just looking for a high-level description of how things import, you should probably skip this section and stick to the main CToSwiftNameTranslation.md.

The main algorithm

At a high level, omit-needless-words takes as inputs:

- a property or method name (including argument names)
- the property type or method result type
- the context ("self") type
- the parameter types, if any
- an optional set of known property names

and produces a new property or method name through the following algorithm:

1. If the result type is the same as the context type, do a leading type name match (see below) against the base name. If the next word *after* the match is a preposition, and there's something following the preposition, drop the matching type name.
    - As a special case, if the preposition is "by" and the following word ends in "ing", drop "by" as well. (Note that the check of the following word is *not* doing full part-of-speech classification as described below; only the first word is classified.)
2. If we're matching a method name rather than a property, do a self type match (see below) against the base name.
3. If we're matching the name of a property or a method with no arguments, and the result type is the same as the context type, do a trailing type name match with special cases (see below) against the base name, then skip to the last step.
4. If the base name starts with "set", do a trailing type name match with special cases (see below) against it, using the context type.
5. If we're matching a method with at least one argument but we don't have a name for the first argument yet, split the base name (see below) to get a new base name and a name for the first argument.
6. If there are parameters and the first argument name is still empty, perform a trailing type name match with special cases (see below) against the base

name, using the first parameter type.

7. Do a trailing type name match with special cases (see below) against each argument name, using the corresponding parameter type.
8. Initialism-lowercase (see below) the first word of the base name and the first word of each argument name.

Part-of-speech classification

In order to make decisions about how to translate method names into Swift, the compiler has a hardcoded list of known prepositions and "base verbs" in lib/Basic/PartsOfSpeech.def. The possible word kinds for Swift are "preposition", "gerund", "verb", and "other".

1. If a word **W** case-insensitively matches a known preposition, it's a preposition. ("within")
2. If a word **W** ends with the characters "ing",
    1. Try dropping the "ing". If the resulting word is a verb, **W** is a gerund. ("reading" matches "read")
    2. Try dropping the "ing" and then adding "e". If the resulting word is a verb, **W** is a gerund. ("coding" matches "code")
    3. Try dropping the "ing". If the last character of the resulting word is the same as the one before, try dropping that too. If the new resulting word is a verb, **W** is a gerund. ("dropping" matches "drop")
3. If a word **W** starts with the characters "auto", "re", or "de", try dropping that prefix. If the resulting word is a verb, **W** is a verb. This process is recursive. ("autoresend" matches "send")
4. Otherwise, **W** does not have a known part of speech and is considered "other".

The hardcoded verb list has several oddities, including a number of British spellings when most of the Apple SDKs use American spellings. "Un-" is also not one of the prefixes checked for when seeing if something is a verb. Unfortunately, changing these rules could break source compatibility.

Initialism-lowercasing

Swift's convention for lowerCamelCased names (like property names, method base names, and argument names) says that the first word should be all-lowercase if it's an initialism. For example, a property that represents a URL could be called "url". Initialism-lowercasing is the transformation that tries to lowercase a possible initialism at the start of a word.

1. If the first character of the name is not an ASCII uppercase letter, no transformation is done.

2. Find the first *non*-uppercase character in the name at position **P**. If no such character was found, the entire string is downcased.
3. If **P** is not the second character in the string, and the character at **P** is an ASCII lowercase letter, and the word starting at **P** isn't plural-suffix-like "s", "es", or "ies", back **P** up by one character. (This assumes the last uppercase character is part of the following word.)
4. Downcase all uppercase letters before **P**.

Examples:

- "sharedApplication" is left unchanged
- "FileManager" becomes "fileManager"
- "URLManager" becomes "urlManager"
- "UTF8String" becomes "utf8String"
- "URLs" becomes "urls"

Property name matching

The compiler sometimes needs to check if a name **N** matches a set of known property names **P**. The algorithm for this is

1. If the first character of **N** is an ASCII uppercase character, and the second character of **N** is not an ASCII uppercase character (or does not exist), the first character of **N** is downcased.
2. If **P** contains **N**, it's a match.
3. If **N** ends in the ASCII character "y", replace that with "ies" and see if **P** contains the result. If so, it's a match.
4. If **N** does not end in the ASCII character "y", append "s" to **N** and see if **P** contains the result. If so, it's a match.
5. If the original **N** does not end in the ASCII character "y", replace the final "s" with "es" and see if **P** contains the result. If so, it's a match.
6. Otherwise, it's not a match.

Type name suffix stripping

Certain suffixes can be stripped from type names for the purposes of matching (see below). The rules for this are fairly straightforward, using the word-splitting algorithm described above:

1. If the last word in a type name **TT** is "Type", "Ref", or "Mask", and that's not the *only* word in **TT**, the last word is dropped.
2. If **TT** ends with the characters "_t", and that's not the *only* characters in **TT**, the "_t" is dropped.
3. If **TT** ends with a series of ASCII digits followed by the character "D", the digits and the "D" are dropped.
4. Otherwise, no stripping occurs.

Examples of stripped prefixes:

- "CFArray**Ref**"
- "intptr**_t**"
- "Point**2D**"

Type name matching

A "type word match" attempts to match a "name word" **N** against a "type word" **T**. Both are assumed to be words from the word-splitting algorithm described above.

1. If **N** and **T** are the same, case-insensitively, they match. ("url" matches "URL")
2. If **N** is a (case-insensitive) suffix of **T**, and the first character of **T** that matches is not a lowercase ASCII character, *and* all leading characters of **T** are not lowercase ASCII characters or underscores, they match. ("url" matches "NSURL")
3. If **N** is a (case-insensitive) prefix of **T**, and all trailing characters of **T** are ASCII digits, they match. ("vector" matches "Vector3")
4. Otherwise, they don't match.

A "type name match" finds the maximum number of type-word-matching words between a series of "name words" **NN** and a series of "type words" **TT**. The search in **NN** can be anchored at the start (a "leading type name match") or the end (a "trailing type name match"); the search in **TT** is anchored at the end in both cases. For a leading type name match:

1. Let **N** be the first word in **NN**.
2. Find the last word **T** in **TT** that **N** type-word-matches. If there is no such word, **NN** and **TT** and 0 is produced.
3. Check that all words in **TT** that follow **T** type-word-match the words in **NN** following **N**. If they do not, **NN** and **TT** do not match and 0 is produced.
4. Otherwise, produce the offset into **NN** that follows all the words that matched **TT**.

Examples of leading type name matches, with the "overlap" in bold:

- "**specialViewController**" matches "My**SpecialViewController**"
- "**viewController**" matches "MySpecial**ViewController**"
- "**viewController**Creator" matches "MySpecial**ViewController**"

A trailing type name match is conceptually the same but has a simpler search strategy:

1. Walk backwards word-by-word in both **NN** and **TT** until you find a mismatch.
2. Produce the offset into **NN** that precedes all of the words that matched **TT** (which might be the entire length of **NN** if no words matched).

Examples of trailing type name matches, with the "overlap" in bold:

- "**specialViewController**" matches "My**SpecialViewController**"
- "**viewController**" matches "MySpecial**ViewController**"
- "parent**ViewController**" matches "MySpecial**ViewController**"

"Trailing type name match with special cases"

In practice, while the leading type name match algorithm is used unaltered, trailing type name matches are performed with a variety of additional special cases. In this case, we consider

- a series of name words **NN**
- a series of type words **TT**
- an optional additional series of *element words* **EE** for a collection element type
- a set of *known property names*
- a *match kind* that is one of the following:
  - method base name: the base name of a method (usually matched against the first argument type)
  - first parameter name: the first parameter of a function or method, unless it has a default argument or the method is an initializer
  - normal parameter name: any other parameter of a function or method
  - property-like name: a property *or* the base name of a no-argument method that returns the "self" type (static or dynamic) *or* the base name of a method whose name begins with "set".
  - collection element: used for matching against **EE** (see below)

In addition to the rules for matching words as described in normal type word matching,

- the word "Indexes" and the word "Indices" in **NN** each match the *two* words "IndexSet" in **TT**.
- the word "Index" in **NN** matches the word "Int" and the word "Integer" in **TT**.
- the two words "ObjectValue" in **NN** match the word "Object" in **TT**.
- if any word in **NN** ends in the character "s", the "s" is dropped and the next part of **NN** is matched against **EE** instead as a "partial" name. If there is a match in **EE** and the next word in **NN** before the match is a preposition, verb, or gerund, the matching part is dropped. ("append**Views**" matches "NSArray<NS**View** *>" and "with**AttributedStringsSet**" matches "NS**Set**<NS**AttributedString** *>")
- if no words have been matched yet, try type name suffix stripping on **TT** (see above) and start over

The general pattern of walking backwards still occurs.

Once a match offset into **NN** has been found, trailing type name matches still go through some additional checks before stripping the matching range. If any of these conditions are true, the name is left alone.

1. The *entire* name matches, and we're not matching a collection element or a "first parameter name".
2. The last word of **NN** is "Error" and that's all that would be stripped.
3. We're matching anything but a property-like name and the word before the match is not a preposition, verb, or gerund.
4. We're matching a method base name, and the word before the match is the first word in **NN**, *and* it's a preposition.
5. We're matching a method base name, and the *matched* text matches a known property name (see above).
6. We're matching a method base name or a property-like name, and the result is a reserved Swift member name (currently "init", "self", "Protocol", and "Type").
7. We're matching a method base name or a property-like name, and the result is "get", "for", "set", "using", or "with" (a "vacuous" result).

If none of these conditions are true, the matching part of **NN** is removed and the remaining string is returned.

Self type matching

Objective-C method names often include a reference to the context type, such as `dismissViewControllerAnimated:`. In Swift, that context type reference is considered superfluous and the result (before base name splitting) should be `dismissAnimated(_:)`. To strip it out, omit-needless-words does a modified trailing type name match with special cases (yes, there's yet another variant of this thing) against the method base name, using the contextual type. What's different:

- There's no collection element matching.
- If there's no initial match, the last word of the name is temporarily stripped off (to be appended again if any stripping occurs) and the search continues. This will happen until a match is found or the compiler runs out of base name.
- The word before the match must be a verb.

Additionally, for every match after stripping the last word of a base name, type suffix stripping is applied as much as possible before *doing any matches. This was probably unintentional.*

Base name splitting

Because Objective-C has selector pieces rather than argument labels, it doesn't have a separate notion of "base name" and "first argument label". Base name splitting is an attempt to bridge that gap.

1. If the parameter is a known boolean type and the last word of the base name is "Animated" (case-sensitive in this case), drop that last word and set the first argument label to "animated".

2. If the first word of the base name is "set", don't do any splitting.

   This probably should have been tested before step #1.

3. If the last word of the parameter type is "Object" (which includes the `id` type) and the *body* parameter name is "sender", assume the method is IBAction-like and don't do any splitting.

4. Search backwards word-by-word through the base name to find the last preposition in the name at position **P**.

   1. If that preposition is "of", search for another preposition that comes before it. If the new search result is not "of" or "for", make that the new **P**.

5. If the preposition at **P** is "in" and the previous word is "plug", give up and don't do any splitting—we found the noun "plug-in" instead.

6. Check the preposition at **P** and the following word. If it is any of the following pairs, give up and don't do any splitting—these don't describe the argument.

   - "with error"
   - "with no"
   - "to visible"
   - "to backing"
   - "from backing"
   - "and return"

7. Check the preposition at **P** and the previous word. If it is any of the following pairs, consider the previous word to be part of the preposition as well and move **P** back one word.

   - "compatible with"
   - "best matching"
   - "according to"
   - "bound by"
   - "separated by"

8. If the rest of the base name following the preposition is "X", "Y", or "Z", let the "X", "Y", or "Z" be the new argument name, and drop it from the base name to form the new base name. Otherwise, let everything before **P** be the new base name, and everything after **P**, including the preposition, be the argument name.

9. If the preposition is "with", and the following word is not "zone", and the parameter is not a function type and does not have a default argument, drop the "with" entirely.

10. If the preposition is “using”, and the parameter is not a function type and does not have a default argument, drop the “using” entirely.
11. If the new base name is a reserved Swift member name (currently “init”, “self”, “Protocol”, and “Type”), don’t do any splitting after all.
12. If the first word of the new base name is “get”, “for”, “set”, “using”, or “with” (a “vacuous” word), and that’s either the entire new name or there’s only one other word, don’t do any splitting after all.
13. Otherwise, return the new base name and the argument name.

Converting an Objective-C type to a type name for matching

1. Strip off any typedefs, except those explicitly mentioned below, as well as any pointer or reference types and any other forms of type sugar (like type attributes).
    1. The typedef “BOOL” gets a type name of “Bool” and a special note that it’s a boolean type.
    2. The typedefs “NSInteger”, “NSUInteger”, and “CGFloat” are preserved as is.
    3. Typedefs for pointers whose names end in “Array” or “Set” have their names preserved; their pointee type goes through this process as well to be used as the “collection element type” (as described in “trailing type name matching with special cases” above).
    4. Typedefs that refer to CF types (see CToSwiftNameTranslation.md) are preserved as is.
2. C array types get a type name of “Array”; their element type goes through this process as well to be used as the “collection element type”.
3. Objective-C selectors (`SEL`) get a type name of “Selector”.
4. An Objective-C object pointer with only a single protocol and a base type of either `id` or `NSObject` gets mapped as the name of that protocol.
5. A generic Objective-C class whose name ends in “Array” or “Set” has its name preserved, even if there are protocol qualifications. Its first generic parameter, if present, goes through this process as well to be used as the “collection element type”. If no generic parameters are present, “Object” is used as the collection element type.
6. A non-generic Objective-C class whose name ends in “Array” or “Set” has its name preserved, even if there are protocol qualifications. Moreover, the same name with the word “Array” or “Set” dropped from the end is used as the “collection element type”.

7. Other Objective-C classes have their names preserved as is, even if there are protocol qualifications on the type.
8. Objective-C `id` gets a type name of "Object", even if it has protocol qualifications.
9. Objective-C `Class` gets a type name of "Class", even if it has protocol qualifications.
10. Tag types (structs, enums, and unions) have their names preserved as is, or their immediately-containing typedef if the tag type itself is anonymous.
11. All block types get a type name of "Block" and a special note that they are function types.
12. All C function types get a type name of "Function" and a special note that they are function types.
13. The built-in types below are mapped as shown:
    - `void` becomes "Void"
    - `float` becomes "Float"
    - `double` becomes "Double"
    - `char8_t` becomes "UInt8"
    - `char16_t` becomes "UInt16"
    - `char32_t` becomes "UnicodeScalar"
14. The built-in `bool` (`_Bool`) type is mapped to "Bool" with a special note that it's a boolean type.
15. All C integer types are mapped to "IntN" or "UIntN" based on their signedness and bit-width, including `char` and `wchar_t`.