

# Rich Text, Core Text

Rob Napier — UIKonf 2020

First, an apology. My description of this talk includes a lot of topics that won't be in this talk. I was going to talk broadly about text layout, and that talk turned out to be about an hour and a half.

# Characters and ©<sup>l</sup>γp<sup>h</sup>ſ

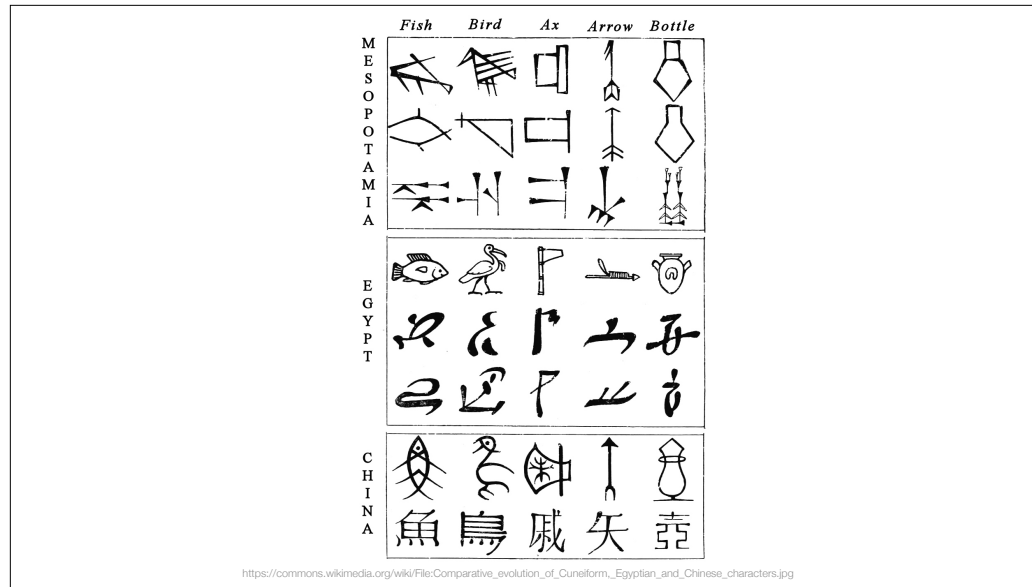
Rob Napier — UIKonf 2020

So I decided to trim it down to just one part of text layout. Characters, fonts, glyphs. That talk was about 45 minutes.

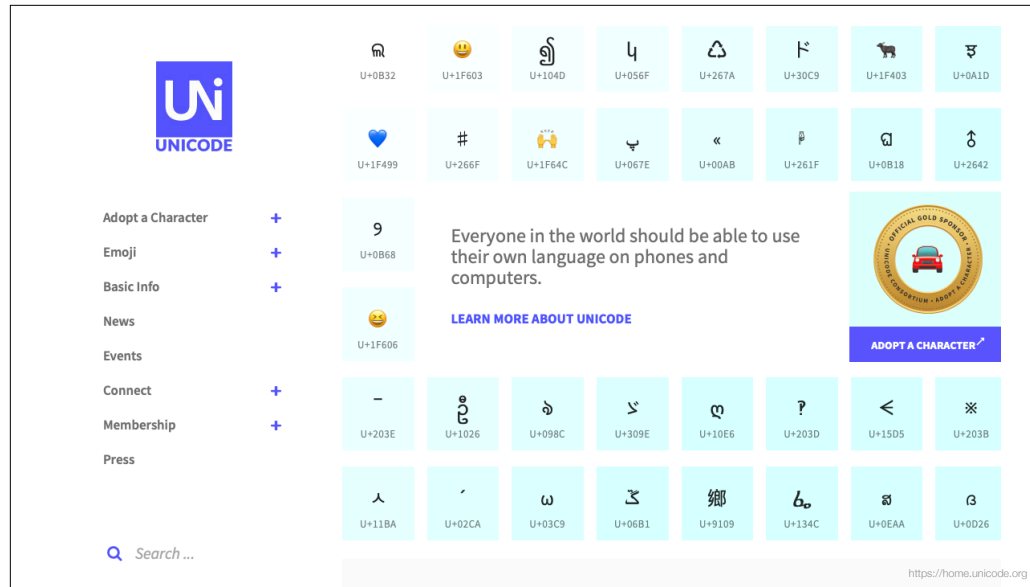
# Just One Part of Unicode

Rob Napier — UIKonf 2020

So, I refocused, and I think that's going to fit into time. So here we go. Just One Part of Unicode.



Humans like to write things down, and over the last several thousand years, we've developed a wide variety of ways to do that. And then, over the last century, we started building computers. And computers work with numbers, which does not line up well with how humans write things down. For decades, we developed numerous systems for encoding human writing into numbers so that computers could work with it.



In the 1980s and early 1990s, Unicode was developed to unify these systems so that there would be a single, consistent way to encode all the world's writing systems, including the ability to include multiple writing systems in the same document. It includes rules for comparing and sorting, text direction, composing and decomposing characters, and on and on. Unicode has made it practical for us to develop software that supports the native writings systems of the whole world. But that is not the most important thing that Unicode has given us. No. The most important thing that Unicode has made possible is...



Emoji.

OK, that's definitely not true. But it feels like. For years, I've been teaching people about bidirectional text and composing characters and normalized forms and why you can't subscript Strings by an integer in Swift, and a common response is:

“Look, I don’t care about all  
these fancy writing systems.  
I just need English.”

“And emoji?”

“Yeah, of course emoji.  
English and emoji.”

“Look, I don’t care about all these fancy writing systems. I just need English.”<build>

And I ask? “And emoji?”<build>

And they say “yeah, of course emoji. English and emoji.”

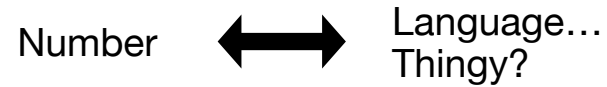


In my experience, emoji is the most complex writing system in the world. It is filled to the brim with weird grammar, special cases, and continual churn. It requires its own complex keyboards. If your code can support emoji on iOS, you get Chinese for free, and most other writing systems, too.

So let's talk about Unicode.



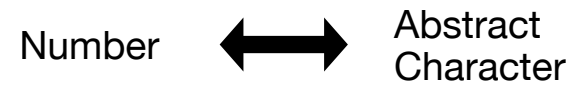
## Code Points



The code point is the most basic unit of text in a computer encoding system. Ultimately, computers work with numbers, so we have to have a way to turn letters and symbols into numbers, and code points are how we do that. In Unicode, code points map a unique number to each “language thingy” for every human writing system.

What do I mean “language thingy?” Well. It’s complicated.

## Code Points



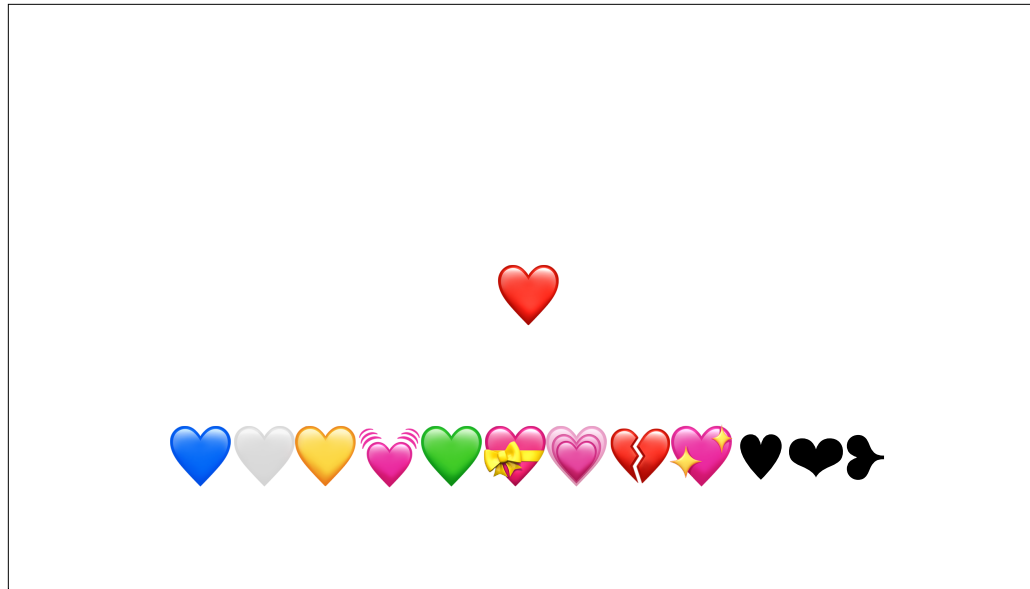
According to Unicode, code points map to “abstract characters.” What does that mean? The intention is that an abstract character is a fundamental unit of written language that’s independent of how it’s drawn.



í

The fact that in most Latin-using cultures, you'd recognize this as a "lowercase x with an accent," even though it's not a letter in any language I've ever heard of. That tells us that "lowercase x" and "with an accent" are probably abstract characters in our culture.

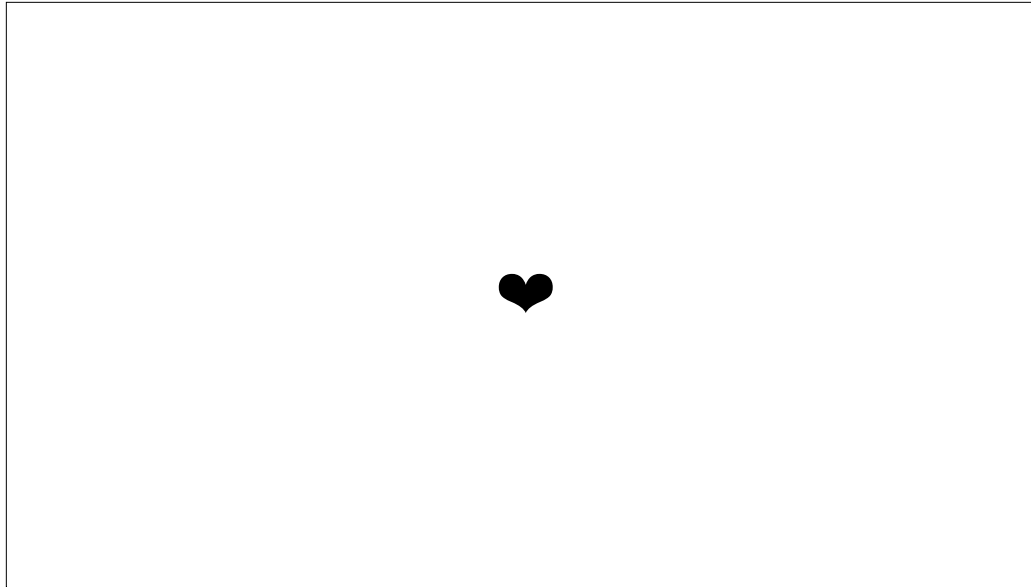
Now Unicode is wildly inconsistent about exactly how this idea is applied, and really there's no universally consistent way you could apply it. Unicode is the way it is for many reasons. Backward compatibility with older systems, global politics, mistakes, and frankly, human writing systems are deeply messy, and so anything that tries to encode them is going to be deeply messy. But the idea is that there are these abstract pieces of human writing, independent of exactly how they're drawn, that each code point represents.



Emoji is no different about this. They have code points, and they have a long history that make their code points inconsistent and messy. A lot of emoji pre-date Unicode and have naming and syntax that come from older systems.

For example, you want to show a red heart. Probably the most common color for hearts to be. So you search the official list of emoji for a red heart, and you don't find one. There's blue, white, yellow, beating, green, with ribbon, growing, broken, sparkling, black, and heavy black. There's even rotated heavy black heart bullet. But where's red? That seems weird.

First, many emoji come from phones that didn't have color screens. So the words "white" and "black" don't always mean the colors specifically, they sometimes mean outlined and filled. And "heavy" means "bold" or "thick."



So this is HEAVY BLACK HEART because it's filled in and wider than BLACK HEART SUIT.

Some Unicode characters can be followed by a variation selector. This is used for things like distinguishing between Chinese-based characters that were merged by Unihan but need to round-trip to other encoding system where they're distinct. But of course no one cares about that. We all use variation selectors for emoji. Every Unicode feature is eventually for emoji.

🖤 + VS15 = 🖤

🖤 + VS16 = ❤️

The two variation selectors used by emoji are 15 and 16. 15 is the “text version” of a character. And 16 is the “emoji version” of a character. So HEAVY BLACK HEART plus VS16 give us an emoji version of filled in, bold heart. Let me show you this in Swift.

```
let redHeart = "❤️"  
  
print(redHeart.unicodeScalars.map { String(format: "%X", $0.value) })  
// ["2764", "FE0F"]  
  
"\u{2764}\u{FE0F}"  
// "❤️"
```

<https://www.fileformat.info/info/unicode/char/fe0f/index.htm>

<https://emojipedia.org>

Say I found this emoji I want to understand, so I copy and paste it into my Playground. To see what it's made of, convert each unicodeScalar, unicode code point, to hex. And then to reconstruct it, use the backslash-u notation with the code point in curly braces. But what if you don't know what FE0F means?

I use my favorite site for looking up characters. [fileformat.info](https://www.fileformat.info).

For emoji-specific information, I like emoji-pedia, but it only covers things related to emoji, while fileformat includes all Unicode.

resumé

U+00E9 LATIN SMALL LETTER E WITH ACUTE

U+0065 LATIN SMALL LETTER E  
+  
U+0301 COMBINING ACUTE ACCENT

This idea of combining code points to make a single character is extremely common in Unicode. Probably the most common example is the accented e in a word like `resumé`. There are two ways in Unicode to express that e. For somewhat historical reasons, it's character U+00E9, LATIN SMALL LETTER E WITH ACUTE. But it can also be encoded as U+0065 LATIN SMALL LETTER E, followed by U+0301, COMBINING ACUTE ACCENT. These are both equally legitimate encodings. So when I want to search for the string "resumé", how does that work?

In Unicode there are standardized ways to compose and decompose characters, called Normalization forms. Swift handles these mostly for you.



```

let e = "e"           // U+0065 LATIN SMALL LETTER E
let acute = "\u{0301}" // U+0301 COMBINING ACUTE ACCENT
let eAcute = "é"      // U+00E9 LATIN SMALL LETTER E WITH ACUTE

e + acute == eAcute   // => true


e.count           // 1
acute.count       // 1
(e + acute).count // 1

var str = "e"      // "e"
str.append(acute)  // "é"
str.removeLast()   // ""
str               // ""

```

For example, these are equal in Swift. And the great thing is if you're working in Swift, you can often ignore all these issues, which is why I'm not going to go into the details, but I want you to be aware of how they can create unintuitive results.<build> For example, string lengths don't always mean what you expect them to mean, and sometimes 1 + 1 is 1. And adding a character and then removing it may not do what you expect. And that's because Swift's idea of characters is pretty close to "thing that takes one unit of space on the line," which is pretty close to what our intuition tends to be, but doesn't map to Unicode code points in any consistent way. You have to be very careful when you start manipulating strings in terms of characters, particularly if you're processing your data in chunks. It may not always do what you expect. You could split a character in half.

```
let girl = "👧"  
let umlaut = "\u{0308}" // U+0308 COMBINING DIAERESIS  
girl + umlaut  
"X" + umlaut  
"x" + umlaut
```



Emoji are characters, so you can apply diacritics like accents on them. But...it doesn't usually work very well. While emoji are fairly standard Unicode characters, they're not standard font glyphs. Normal characters are stored as vector graphics, and there are various metrics that the text layout system uses to place combining marks correctly, or at least reasonably. Notice how the umlaut is well-placed for both the capital and lowercase x, even though "x with an umlaut" is not something this font would have baked in like accented e. Emoji are stored in a font file as PNGs at various resolutions, and the text system doesn't know where to put combining marks, so they get placed kind of randomly.



As long as we're talking about combining characters, I might as well take a brief detour into a fun way to abuse them. Zalgo text. If you've ever seen this kind of weird "corrupted" text before, it's just using Unicode.

```

func zalgo(_ text: String) → String {
    let combining = (0x0300 ... 0x036f) // The main block of combining characters
    .compactMap(UnicodeScalar.init) // Converted to Characters
    .map(Character.init)

    // For each character
    return text.map { char in
        // Pick a random number of characters to add
        let zalgoChars = (0 ..<Int.random(in: 0 ... 15))
        .map { _ in combining.randomElement()! }
        return String(char) + String(zalgoChars)
    }.joined()
}

```

Zâlgô!

There are a lot of possible combining characters in Unicode, but there's a large block of them that work well and are supported on most platforms in the 0300 page. So for each character in the original text, you pick between zero and 15 combining characters, and add them between each character. But what if we want to de-zalgo this text? Remove all the combining characters. Of course we could just look for characters in the 0x0300 range, but we can also ask Swift for information about each character.

```
func dezalgo(_ text: String) → String {  
    // For each character in the string  
    text.map { char in  
        // Make a new String  
        String(  
            // By filtering the code points  
            char.unicodeScalars.filter { scalar in  
                // To ones that are base graphemes  
                scalar.properties.isGraphemeBase  
            })  
        }.joined()  
    }  
  
    dezalgo(zalgo("Zalgo!"))    // Zalgo!
```

In Swift, a String is made of Characters. And a Character is made of code points, or Unicode Scalars. Each Unicode Scalar, each code point, has a lot of information about what kind of thing it represents. You can get that by checking its properties. There's tons of these. You can check if it's a letter or a number or emoji, or marks the end of a sentence. You can even find out its numeric value for things like the 1/5 fraction or 4 in a circle. Most of the time you probably want to use methods on Character rather than diving all the way down into the UnicodeScalar properties, but sometimes it's useful for filtering out things like this.

## Emoji Combining Characters





```
let keycap = "\u{20E3}" // U+20E3 COMBINING ENCLOSING KEYCAP  
"1" + keycap  
  
let var16 = "\u{FE0F}" // U+FE0F VARIATION SELECTOR-16  
"1" + keycap + var16
```



Emoji have their own systems of combining to form a single character. I've already shown how variation selectors work, but there are a lot more.

For example, you can put a box around numbers by adding COMBINING ENCLOSING KEYCAP. But you can also give them an emoji appearance by adding the emoji variation, 16.

## Flags - Regions

<code>let regionU = "\u{1F1FA}"</code>	//	<code>U</code>	<code>regionU + regionS</code>	//	
<code>let regionS = "\u{1F1F8}"</code>	//	<code>S</code>	<code>regionD + regionE</code>	//	
<code>let regionD = "\u{1F1E9}"</code>	//	<code>D</code>	<code>regionU + regionN</code>	//	
<code>let regionE = "\u{1F1EA}"</code>	//	<code>E</code>	<code>regionE + regionU</code>	//	
<code>let regionN = "\u{1F1F3}"</code>	//	<code>N</code>	<code>regionN + regionN</code>	//	<code>NN</code>

Flags are composed of two regional letters. These aren't normal letters, they're code points specifically for this purpose. They match ISO region codes.

**“Some region sequences represent countries (as recognized by the United Nations, for example); others represent territories that are associated with a country. Such territories may have flags of their own, or may use the flag of the country with which they are associated. Depictions of images for flags may be subject to constraints by the administration of that region.”**

Unicode Technical Standard #51, *Unicode Emoji*, Annex B: “Valid Emoji Flag Sequences.”

The spec would make clear is not a statement of whether something is or is not a country. Some are territories, some disputed territories, and the ISO committee would really like to stay out of global politics about it. That of course is impossible. But bless their hearts, they try.



## Flags - Regions

```
let regionU = "\u{1F1FA}" // [U]
let regionS = "\u{1F1F8}" // [S]
let regionD = "\u{1F1E9}" // [D]
let regionE = "\u{1F1EA}" // [E]
let regionN = "\u{1F1F3}" // [N]

regionU + regionS // 🇺🇸
regionD + regionE // 🇩🇪
regionU + regionN // 🇺🇳
regionE + regionU // 🇪🇺
regionN + regionN // [N][N]

let zwsp = "\u{200B}"
regionU + zwsp + regionS // [U][S]
```

There are two macro-regions, the UN and EU, but they act just like regular regions, so it's not a big difference.

If the letters don't form a known region, then they're displayed in these dashed boxes. Sometimes people want to use these characters for the dashed box effect itself, but of course that's tricky, because it might form an identifier.<build>

Now I'd never tell you to abuse characters for things they weren't intended for, and the region characters are for regions, not for fancy display purposes. But it happens to be true that you can put zero-width white space between them, and that is a general tool for avoiding character combinations that you don't want.

## Flags - Emoji Tag Sequences

```
let blackFlag = "\u{1F3F4}" // 🚩
let tag_g = "\u{E0067}"
let tag_b = "\u{E0062}"
let tag_s = "\u{E0073}"
let tag_c = "\u{E0063}"
let tag_t = "\u{E0074}"
let tag_cancel = "\u{E007F}"

blackFlag + tag_g + tag_b + tag_s + tag_c + tag_t + tag_cancel // 🇬🇧🇸🇪
```

For regional subdivisions like states, there's a completely different system, which is tags. Tags are a general purpose tool for modifying emoji, but this is the only thing they're currently used for. You start with an emoji, in this case a black flag, it must be black, and then you spell out the tag using the tag letters, again these are not normal letters, they're specifically for tags, and you terminate the string with a tag cancel. For regional subdivisions, the tags are lowercase tag letters that start with the region identifier, followed by the subregion identifier. So for example, the California flag would be u, s, c, a, cancel. But that won't work on most platforms today. The current recommendation only includes subdivisions for Great Britain. So there are flags for English, Scotland, and Wales, and that's it.

I do kind of expect the recommendation to expand to include more regions, particularly the states in the US, but we'll see. You'll notice that tag letters are invisible, and if the tag is unrecognized, it'll generally just display a black flag. The recommendations suggest adding a little question mark to the flag to make it clear that the tag wasn't understood, but Apple doesn't do that currently.

## More Flags!

```
let whiteFlag = "\u{1F3F3}" // 🚩
let skullAndBones = "\u{2620}" // 💀
let rainbow = "\u{1F308}" // 🌈
let transgender = "\u{26A7}" // 🏳️

blackFlag + zwj + skullAndBones + var16 // 🏴‍☠️
whiteFlag + var16 + zwj + rainbow // 🏳️‍🌈
whiteFlag + var16 + zwj + transgender // (coming soon)
```

And of course no discussion of flags would be complete without the pirate flag and the rainbow flag. And this year the recommendation added the transgender flag, but Apple doesn't support it quite yet. I expect soon.

## Not all flags wave

"\u{2691}" // 🚩 "BLACK FLAG"  
"\u{1F3F4}" // 🚩 "WAVING BLACK FLAG"

Unicode	CLDR
WAVING BLACK FLAG	black flag
MOTHER CHRISTMAS	Mrs. Claus
GRINNING FACE WITH STAR EYES	star-struck

Note that the white and black flags here are the WAVING WHITE FLAG and WAVING BLACK FLAG. Not to be confused with WHITE FLAG (U+2690) and BLACK FLAG (U+2691), which won't work for these uses. And unfortunately, if you use the "Show Emoji and Symbols" pane on your Mac, and search for "black flag," you'll get the wrong one.

The other problem is, if you read the spec, it says "black flag," not "waving black flag." Why?<build>

Emoji are usually described in terms of Common Language Data Repository (CLDR) short names, which are generally lowercase, rather than all caps Unicode names. The Unicode names are unique, immutable identifiers, and sometimes are quite long. The CLDR short names are not promised to be unique, can change over time, and can be localized. Most of the time it's not a big deal. The two are usually pretty close in English, but not always, and sometimes it's a little ambiguous like black flag versus WAVING BLACK FLAG.

## Skin tone modifiers

```
let woman = "👩"  
let tone = ["🏠", "🏡", "🏢", "🏣", "🏤"]  
woman + tone[4]
```



```
"\u{1F3FB}" // EMOJI MODIFIER FITZPATRICK TYPE-1-2  
"\u{1F3FC}" // EMOJI MODIFIER FITZPATRICK TYPE-3  
"\u{1F3FD}" // EMOJI MODIFIER FITZPATRICK TYPE-4  
"\u{1F3FE}" // EMOJI MODIFIER FITZPATRICK TYPE-5  
"\u{1F3FF}" // EMOJI MODIFIER FITZPATRICK TYPE-6
```

Let's talk about people. A lot of emoji rules revolve around people, which is understandable, since people are why we have emoji.

Most human emoji accept a skin tone modifier. So for example, WOMAN followed by EMOJI MODIFIER FITZPATRICK TYPE-5. Notice the array of skin tones here. The Fitzpatrick modifiers are rendered as tonal squares they stand alone, unlike the variation selector, which is invisible if it's not modifying anything. This is a bit of a forward and backward compatibility feature. When you apply a skin tone to an emoji that the rendering engine doesn't support skin tone for, it renders separately.

## Skin tone modifiers

```
let family = "👨👩"  
family + tone[3]
```



```
(family + tone[3]).count // 1
```

For example, family emoji don't accept skin tone. I'll talk more about that shortly, but they might some day. Today, this is how they'll render. Note this is still one "character." If you press backspace, both blocks will be deleted together. This is really about forward compatibility. The idea is that if your device doesn't know how to render skin tone on this emoji, then at least the reader can see it.

## Gender

```
let female = "♀" // U+2640 FEMALE SIGN
let male = "♂" // U+2642 MALE SIGN

let zwj = "\u{200D}" // U+200D ZERO WIDTH JOINER
let runner = "\u{1F3C3}" // 🏃
runner + tone[3] + zwj + female // 🏃🏻♀️
runner + zwj + male // 🏃🏻♂️
```

Most human emoji accept a gender modifier as well, which in Unicode means either the FEMALE SIGN or MALE SIGN. But they don't work quite the same way as skin tone modifiers. To act as a modifier, they need to be part of a Zero Width Joiner sequence.<build>

The ZWJ is a character that indicates that the surrounding code points should be connected, even though they wouldn't normally be. It's mostly used in Indic scripts, which are writing systems I haven't studied enough to explain, but they're also very common in emoji to create Zero Width Joiner Sequences like this one. Notice that skin tone modifiers attach to the person, the runner in this case, and then the gender modifier is added with a Zero Width Joiner.

You'll notice that the base RUNNER emoji is not the same as the male RUNNER emoji. The intention is that without a modifier most emoji is gender neutral. There are a few exceptions, but most of them will disappear in the next version of Unicode.

## Professions

```
let adult = "\u{1F9D1}" // U+1F9D1 ADULT
let rocket = "\u{1F680}" // U+1F680 ROCKET

adult + zwj + rocket // 🧑🚀

man + tone[2] + zwj + rocket // 🧑🚀
woman + tone[0] + zwj + rocket // 🧑🚀

let baby = "\u{1F476}" // 🧒🚀
baby + zwj + rocket
```

Professions work the other way. The person comes first, then a noun that describes the profession. For example, an astronaut is an ADULT ROCKET. In case you're curious, children cannot have professions. There is no BABY ROCKET. <build>I'm sorry.



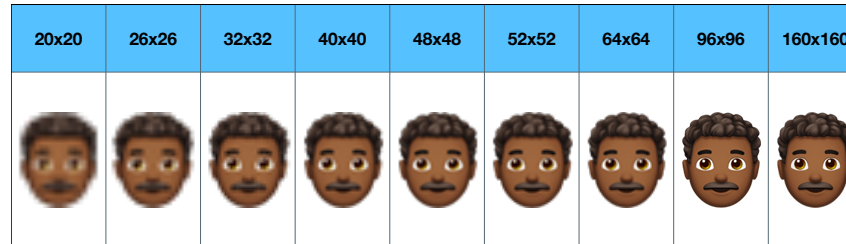
# Hair

```
let redHair = "\u{1F9B0}"
let curlyHair = "\u{1F9B1}"
let bald = "\u{1F9B2}"
let whiteHair = "\u{1F9B3}"

adult + tone[1] + zwj + redHair // 🧑🏻
man + tone[3] + zwj + curlyHair // 🧑🏿
woman + tone[2] + zwj + bald // 🧑🏼
woman + tone[4] + zwj + whiteHair // 🧑🏾
```

Unicode 11 added hair components. There's just four, red, curly, bald, and white. And they only apply to adults, men, and women. Children and babies have no hair. But as you can see, the number of glyphs required grows very fast as we add more and more modifiers. These are each a separate PNG in the font file, sometimes at several resolutions. They're not constructed by the text layout system. So the size of emojis have been growing very quickly.

`man + tone[3] + zwj + curlyHair` // 



Apple Color Emoji	~200 MB
Ping Fang	~75 MB
San Francisco	~0.15 MB

There are currently 3,165 glyphs in Apple's emoji font, each at 9 different resolutions. So that's over 27,000 PNGs. Which is why Apple's emoji font is over twice as large as the main Chinese font which has 48,000 glyphs in it, and over a thousand times larger than San Francisco. This really isn't scalable.

**“The longer-term goal for implementations should be to support embedded graphics, in addition to the emoji characters. Embedded graphics allow arbitrary emoji symbols, and are not dependent on additional Unicode encoding. Some examples of this are found in Skype and LINE.”**

Unicode Technical Standard #51, *Unicode Emoji*, Section 8: “Longer Term Solutions.”

And the Unicode committee knows this. People want to express themselves as their full selves, and our writing systems should empower that. Having each vendor develop its own set of glyphs for every possible combination of human features is getting in the way of that, despite the rather amazing work that has gone into it so far.

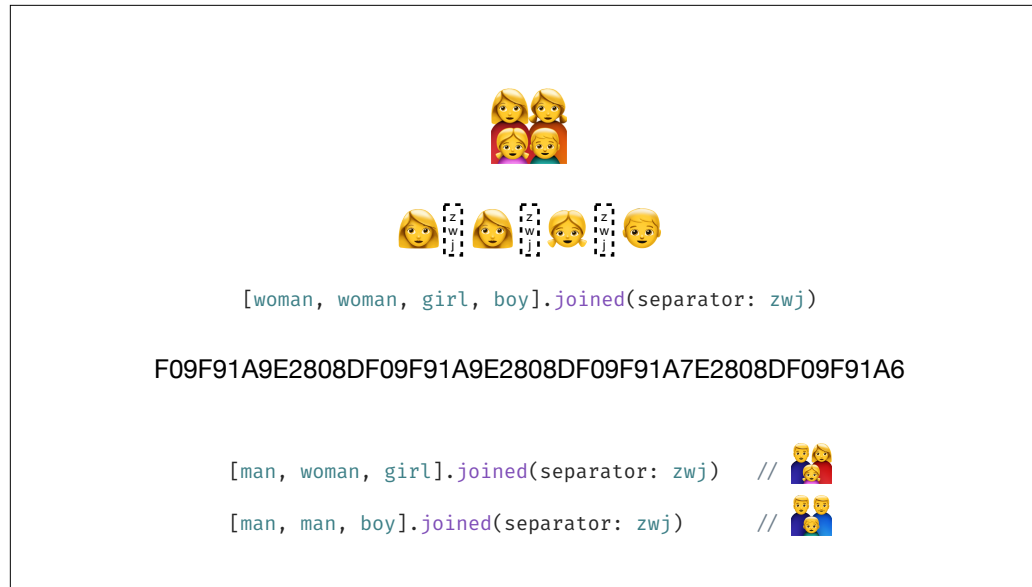
## (Future) Directions

```
let right = "\u{27A1}" + vs16 // ➡️
runner + zwj + right          // 🏃➡️
// Someday: 🏃➡️🏃

let cat = "\u{1F408}"
let black = "\u{2B1B}"
cat + zwj + black             // 🐱🖤
// Someday: 🐱🖤
```

So that said, emoji are getting some new features, that are going to add even more combinations. This doesn't work today on Apple platforms, but hopefully it will soon. You can add an emoji-style left or right arrow to change their direction. But like I said, today it'll show the fallback display of just gluing the two symbols together. Notice the Variation Selector there. The arrow is put into emoji style rather than being a simple text arrow. And that's I think because of fallbacks. You want it to be emoji-like if it's going to be part of an emoji.<build>

Same thing for color. In the future, you should be able to apply color to some emoji like a black cat. Emoji keeps expanding.



And that brings us to my favorite collection of emoji. The multi-person zero-width joiner sequences. For example, this is a family sequence.

This is a single “character.” And that makes sense. Look at it. It’s all in one little square box.<build>

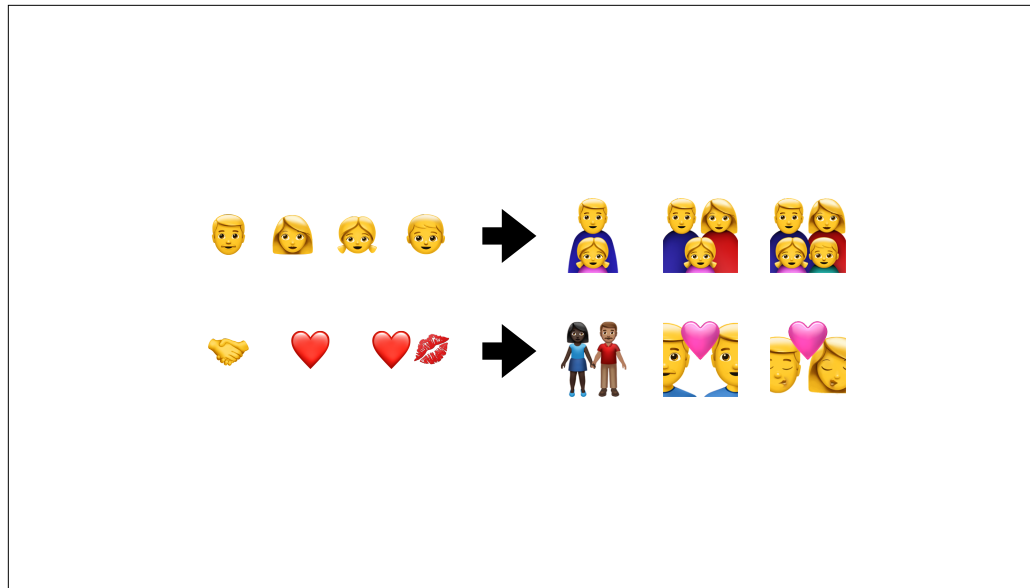
But that’s seven Unicode code points, and in UTF-8, it’s 25 bytes long.

The various other family emojis are created the same way. <build>

man, joiner, woman, joiner, girl. man, joiner, man, joiner, boy, etc.

There’s 25 recommended family combinations in the standard.

Order matters. For example, in families, the order is man, woman, girl, boy. You can have zero, one, or two of each of them, as long as there is at least one parent and one child, but they have to be in that order.



There are many more. People can be joined into a family of two to four people, they can be holding hands, can have a heart between them, can be kissing, they can wrestle, they can stand together wearing bunny ears. Individual skin tone modifiers can be applied to each person when they hold hands, but not in any of the other combinations.

To handle the combinations of skin tone and gender for “two people holding hands” requires 86 unique glyphs, and that doesn’t include the possibility of hair color modifiers, which are also a thing. As the spec notes, adding individual skin tone to family sequences would add over 4,000 new glyphs, so while the spec supports it, I doubt that’ll ever be in the recommendation. And, of course, order matters again. Some multi-person groups support multiple genders, some do not. For non-family sequences that support multiple genders, woman is before man. For families, man is before woman, girls are before boys. And of course there’s also a dedicated “family” emoji code point, separate from all of this. This is the grammar of emoji.

“Look, I don’t care about all  
these fancy writing systems.  
I just need English.”

“And emoji?”

“Yeah, of course emoji.  
English and emoji.”

“I don’t care about localization, I just want English and emoji.”







<https://www.unicode.org/reports/tr51/>

[rob@neverwood.org](mailto:rob@neverwood.org)  
[robnapier.net](http://robnapier.net)  
[@cocoaphony](#)

But I love emoji. I really do. I love all the ways humans have invented to express themselves to each other. Emoji is messy and complicated and sometimes makes you want to scream a bit. Just like people. And like people, they make you smile. I hope you learned something today, and even more I hope I could make you smile, too. Have a great conference.