Nutrient

Rust for iOS Developers

More Swifty than Swift

Douglas Hill
Nutrient

iOSDevUK
9 September 2026

- Nutrient

NSLondon.com

For many years I’ve been helping organise NSLondon.

It’s a wonderful community of iOS and Swift developers meeting every couple of months in London.

Nutrient

Nutrient.io

(PSPDFKit)

The PDF SDK you're looking for

I work at Nutrient. We used to be called PSPDFKit. We make infrastructure for document workflows for agents and humans. We have SDKs for Swift and other languages so you can drop a fully feature PDF viewer and editor into your app.

This year at work, I had the opportunity to move between teams. I currently work mostly with our Server team where we use Elixir.

Outside of work, I also tried other programming languages.

Logo from github.com/rust-lang/rust-artwork

Of the various things I've been learning, the Rust programming language really appealed to me. I’m here today because I thought you might like it too.

I've been using Rust for side projects so far. I wouldn't say I'm a Rust expert. I’m still learning, and I’m here to share my experience.

I’m not going to teach you Rust in this talk.

I’ll start with a bit about Rust at a high level, and why it’s an interesting fit for iOS developers.

We’ll move on to a Rust and Swift language comparison, looking at examples that show different design trade-offs.

I’ll spend more time on memory management since this is one of the tougher concepts with Rust.

Why learn other programming languages?

I’ve been an iOS developer since 2011, and recently I wanted to diversify and learn a lot more.

AI is lowering the barrier for us to apply our skills to other platforms.

But understanding still matters for long-term productivity and quality.

Nutrient

Why Rust?

Mozilla sponsored the initial development of Rust.

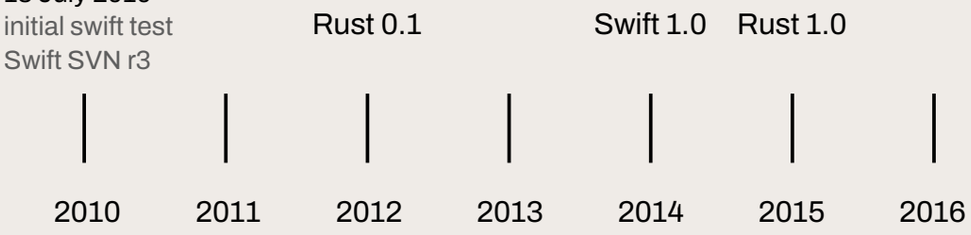
They wanted something with safer memory handling than C++, but still with really good efficiency.

Nutrient

Rust and Swift timeline

`c01efc6`
Graydon Hoare
16 June 2010
Initial git commit.

`18844bc`
Chris Lattner
18 July 2010
initial swift test
Swift SVN r3

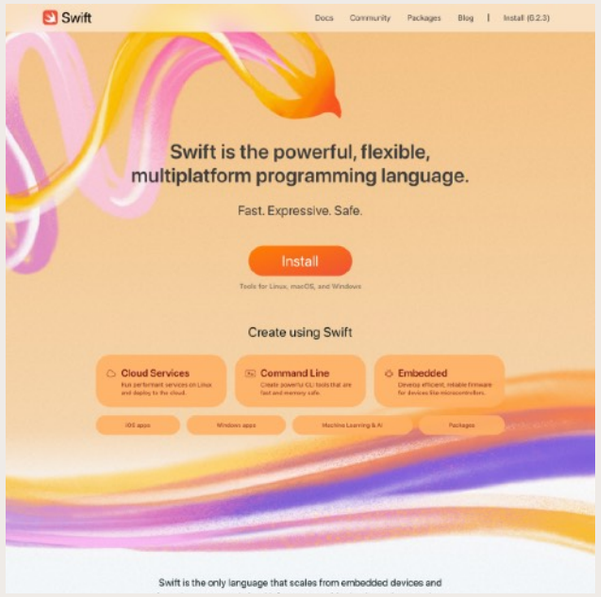


The first commits in the Swift and Rust repos were one month apart.

Version 1 of each came out in 2014 and 2015.

So the same generation of programming language.

- swift.org

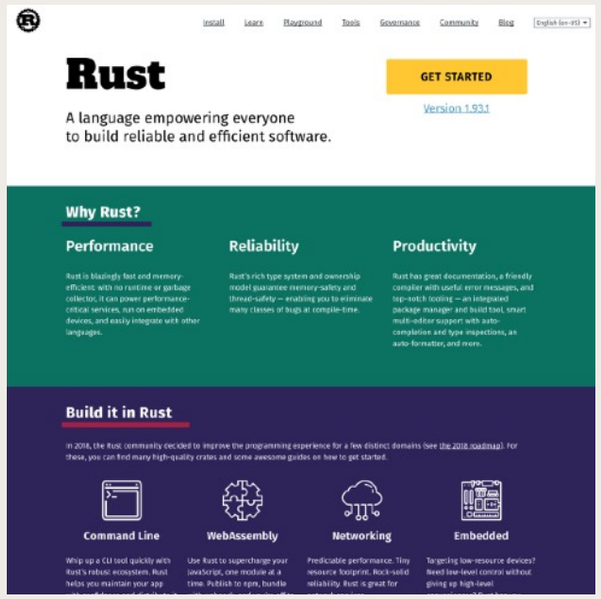


Let’s look at the goals of each language.

The Swift website says: fast expressive safe.

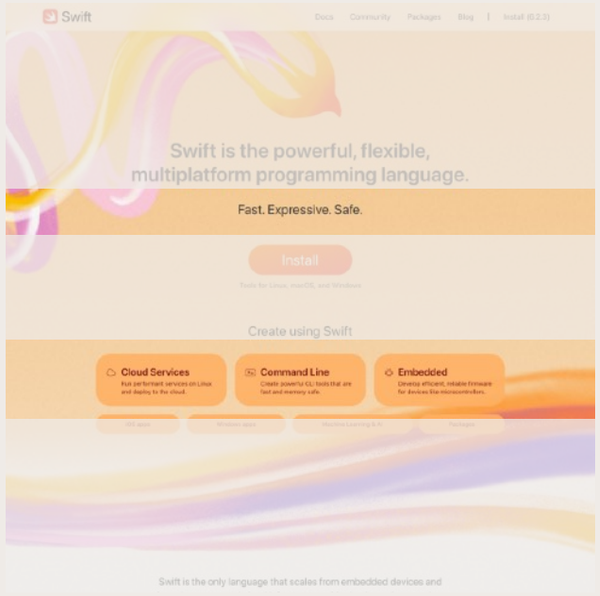
It doesn’t highlight iOS apps much. Instead it’s promoting use for servers, command-line tools and embedded.

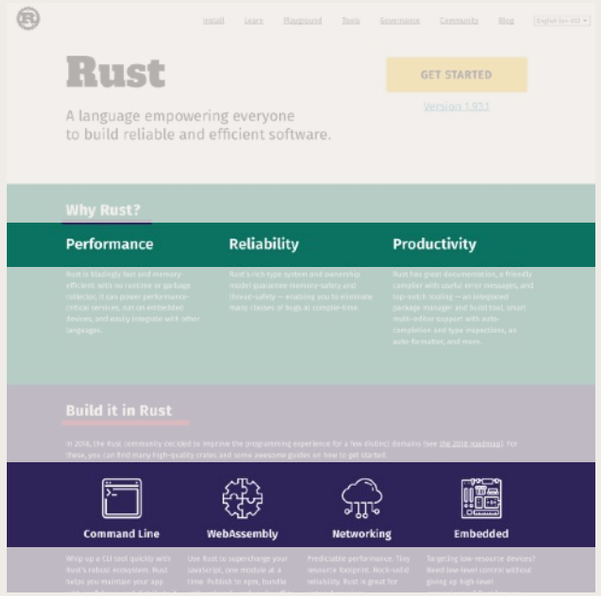
- rust-lang.org



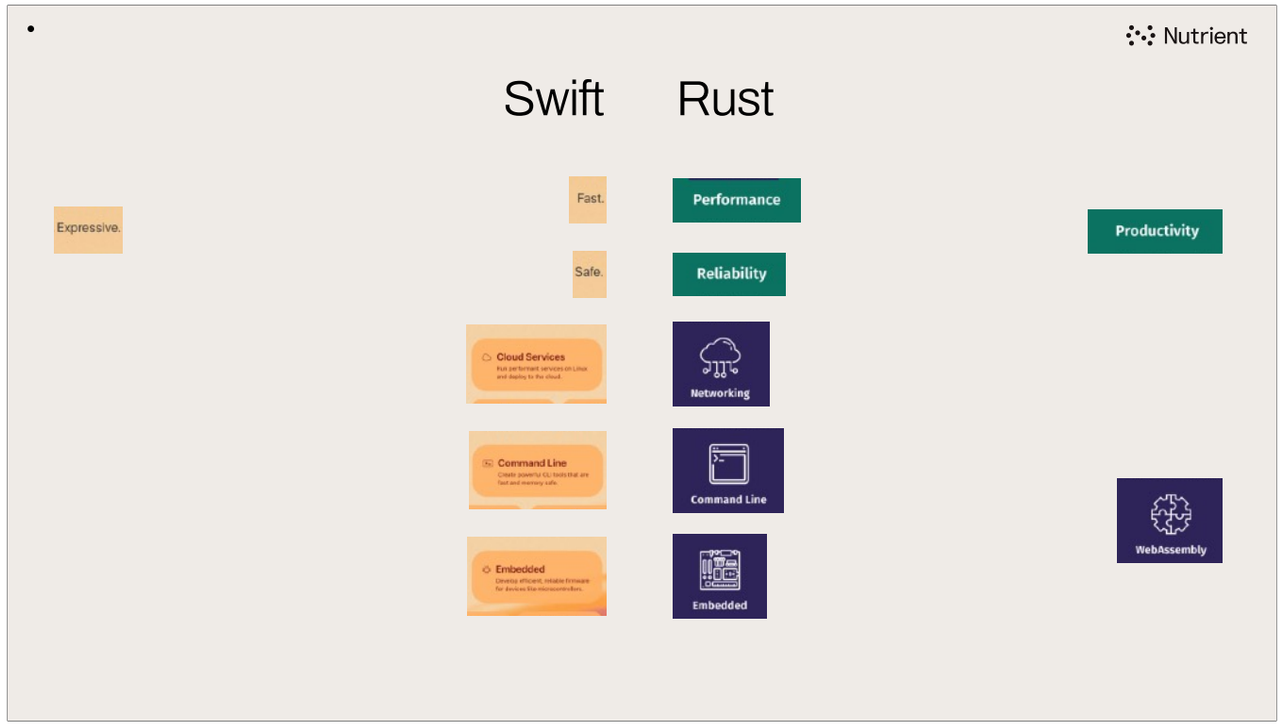
Rust is aiming for performance, reliability, productivity

The highlighted uses are Servers, command-line tools, embedded and WebAssembly.





You may notice…



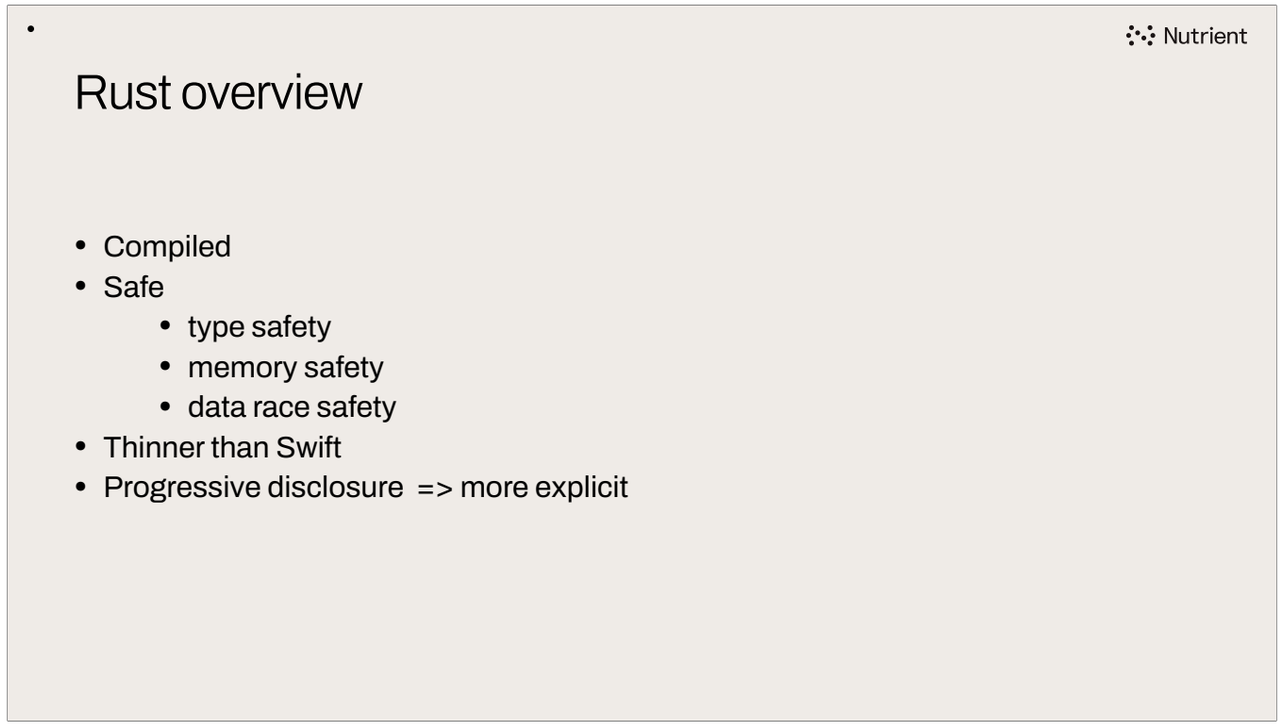
…these look very similar!

Nutrient

Rust is similar to Swift

It's not surprising that similar goals and similar timelines results in languages with interesting similarities.

Rust is a good language to learn coming from Swift because they are so similar. As a consequence of being similar, learning Rust from Swift is not that hard.



Both Swift and Rust compile the old-fashioned way to native machine code. Neither need a virtual machine or interpreter to run.

Both have a strong emphasis on type safety, memory safety, and data race safety.

Rust is a thinner language. It pushes more functionality out of the language into libraries and tooling.

Swift aims for progressive disclosure. You can get more insight into these design choices from Doug Gregor’s talk from SwiftCraft last year.

On the other hand, Rust tends to be more explicit so there’s less hidden magic.

Nutrient

Rust ecosystem

- Packages => **crates**
- Swift Package Manager => **Cargo**
- Swift Package Index => **crates.io** / **docs.rs**

• Both have package managers
• Rust calls packages crates.
• The Rust package manager is called Cargo. People really like Cargo. It's simple to use, but it's also extensible.

• They also both have a centralised place to discover packages.
• Swift Package Index became an official part of the Swift project at Apple. Plan is for it to expand and become a registry.
• This will give Swift Package Index similar functionality to crates.io: Rust’s centralised registry.

Nutrient

Rust is more established outside of iOS development

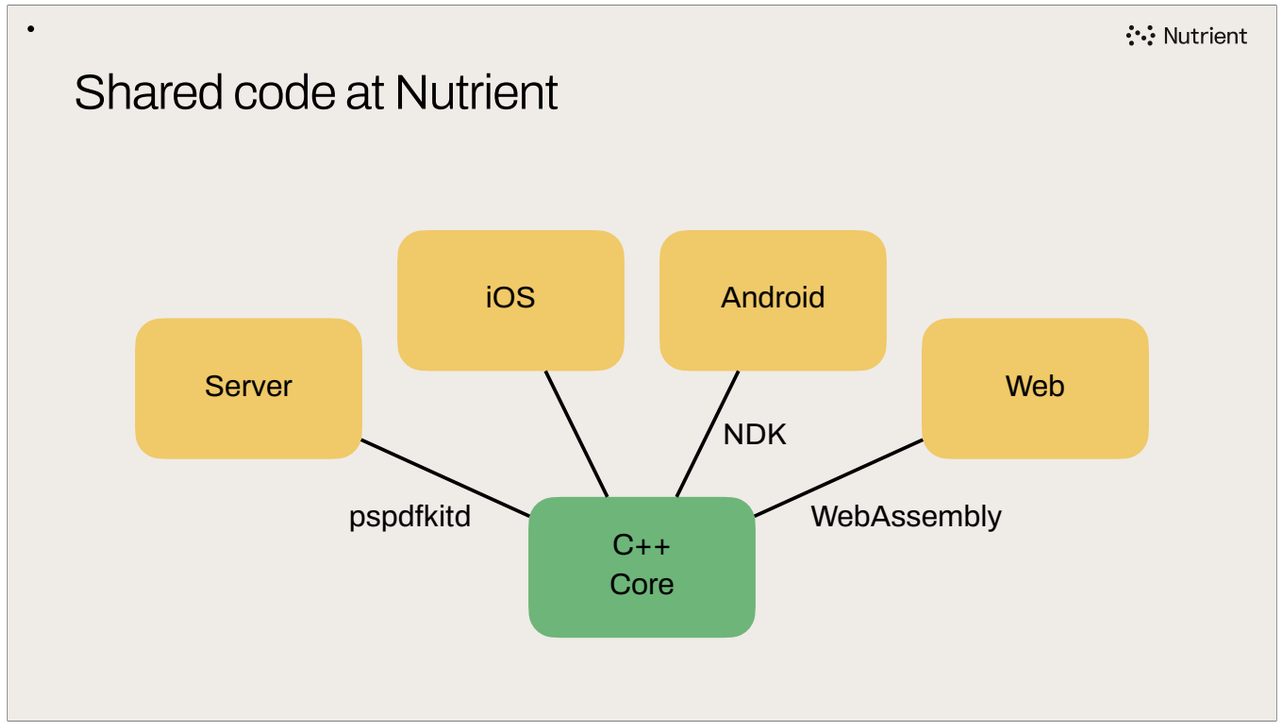
So Swift and Rust are similar, but Rust has a substantially broader established ecosystem in domains outside of iOS development, especially servers, command line and embedded

You'll find libraries for these areas are more mature.

Nutrient

Rust on iOS?

I haven’t looked much at using Rust in an iOS app. Use the best tool for each job. You could use Rust to share code with other platforms.



At Nutrient we share lower-level code using C++. It’s very stable and mature, but Rust would be nicer to work with. It would be more familiar to Swift and Kotlin developers

Nutrient

Language comparison

Nutrient

Similarities between Swift and Rust

`let` and `var`	✔ `var` is `mut let`
Type inference or explicit types	✔
Tuples	✔ no named members
`if`, `else`, `while`, `for`	✔ also `loop`
Exhaustive `switch`	✔ called `match`
Structs	✔ have update syntax
Enums	✔
Protocols	✔ called traits
Generics	✔
`Optional`	✔ called `Option`, no `nil` shorthand
Closures	✔ use |

A lot is similar. Emphasis on immutability with let. Type inference.

We love our enums with associated values in Swift, and we can do the same things in Rust.

Protocols are called traits.

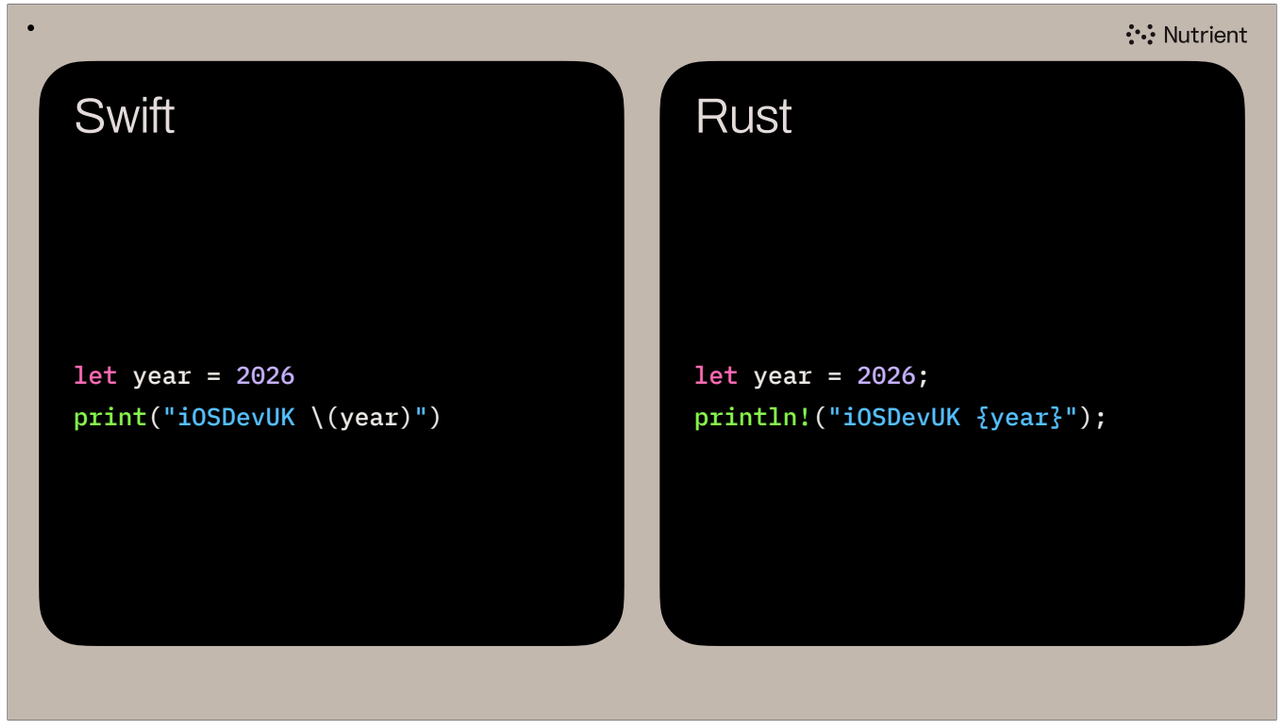
Nutrient

Macros

Let’s look at a few interesting comparisons between the languages.

Similar to Swift, macros are like compiler extensions. There are specific types of macros you can use.

Macros are very commonly used in Rust.

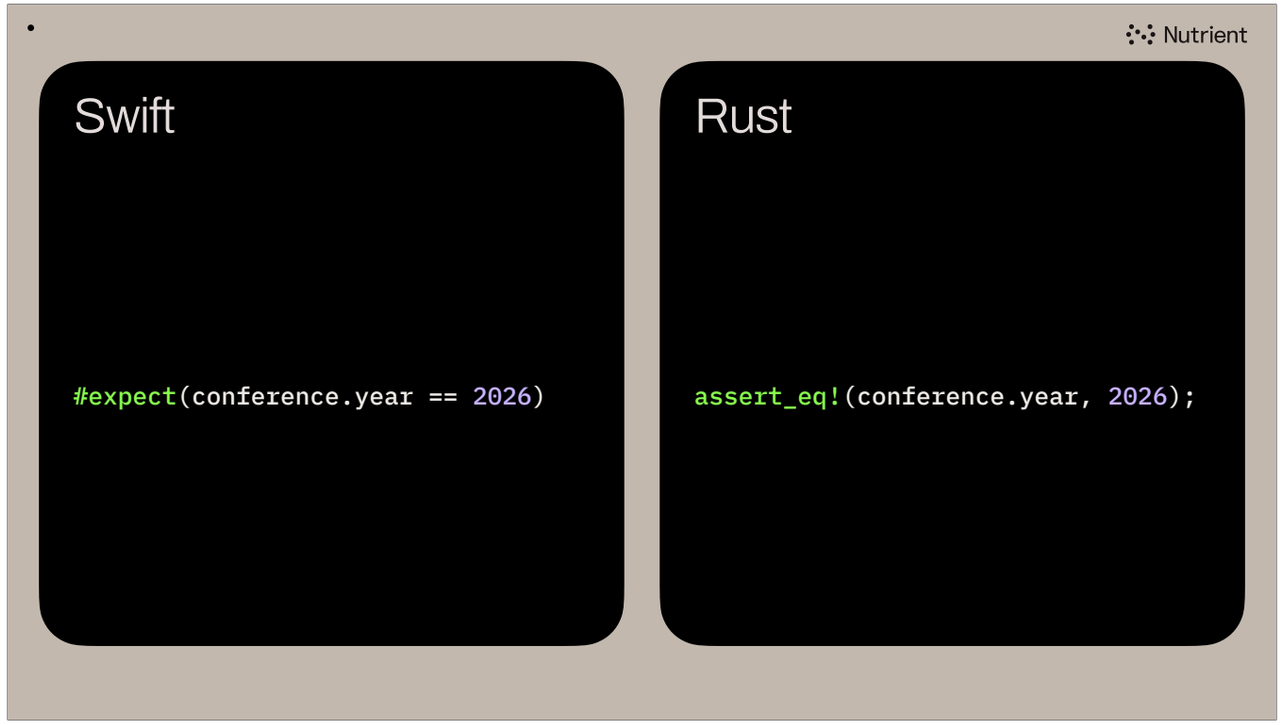


In Swift, `print` is just a function. This relies on string interpolation and variadic functions.

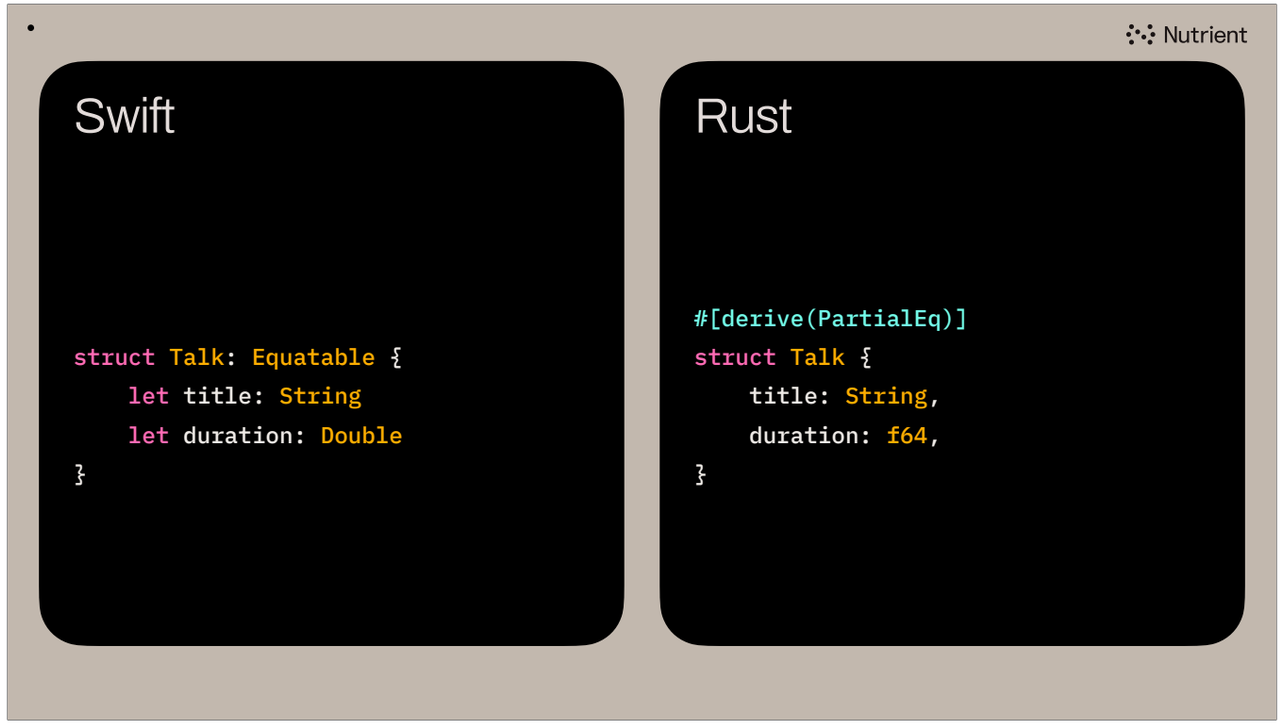
Rust doesn’t have these language features.

Instead, `println!` in Rust is a function-like macro, which is indicated by the exclamation mark.

It’s the macro that does validation of the string formatting at compile time. Checking the number of arguments and that they can all be converted to strings.



We also have function-like macros in Swift, like `#expect` from Swift testing.



In Swift, there are a few protocols where if you conform a type to the protocol, the implementation will be synthesised. For example, Equatable, Hashable, Codable.

These are hardcoded into the Swift compiler.

This isn’t Rust’s style. It’s more explicit and extensible. You use derive macros, which add trait conformances and synthesised implementations. These are really common.

Nutrient

Swift

```swift
@Observable class Talk {
    let title: String
    let duration: Double
}
```

Rust

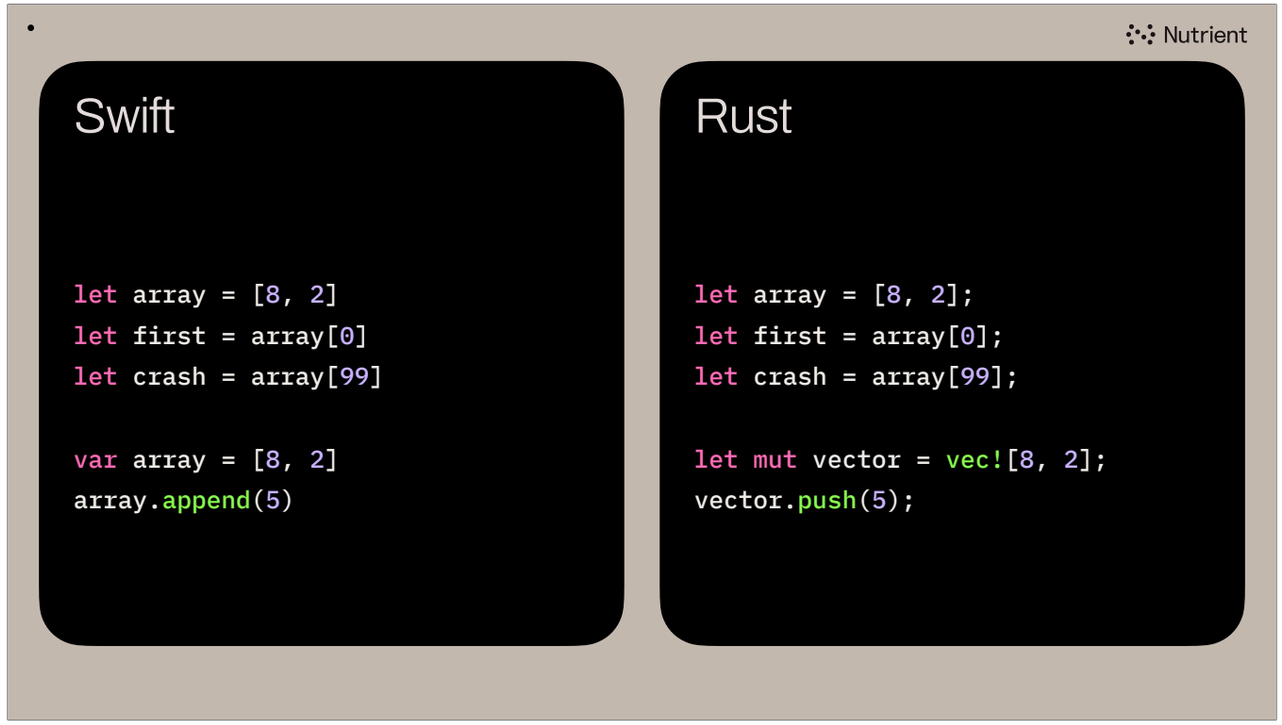
```rust
#[derive(PartialEq)]
struct Talk {
    title: String,
    duration: f64,
}
```

Some newer things in Swift add protocol conformances using macros, like `@Observable`.

Nutrient

Arrays

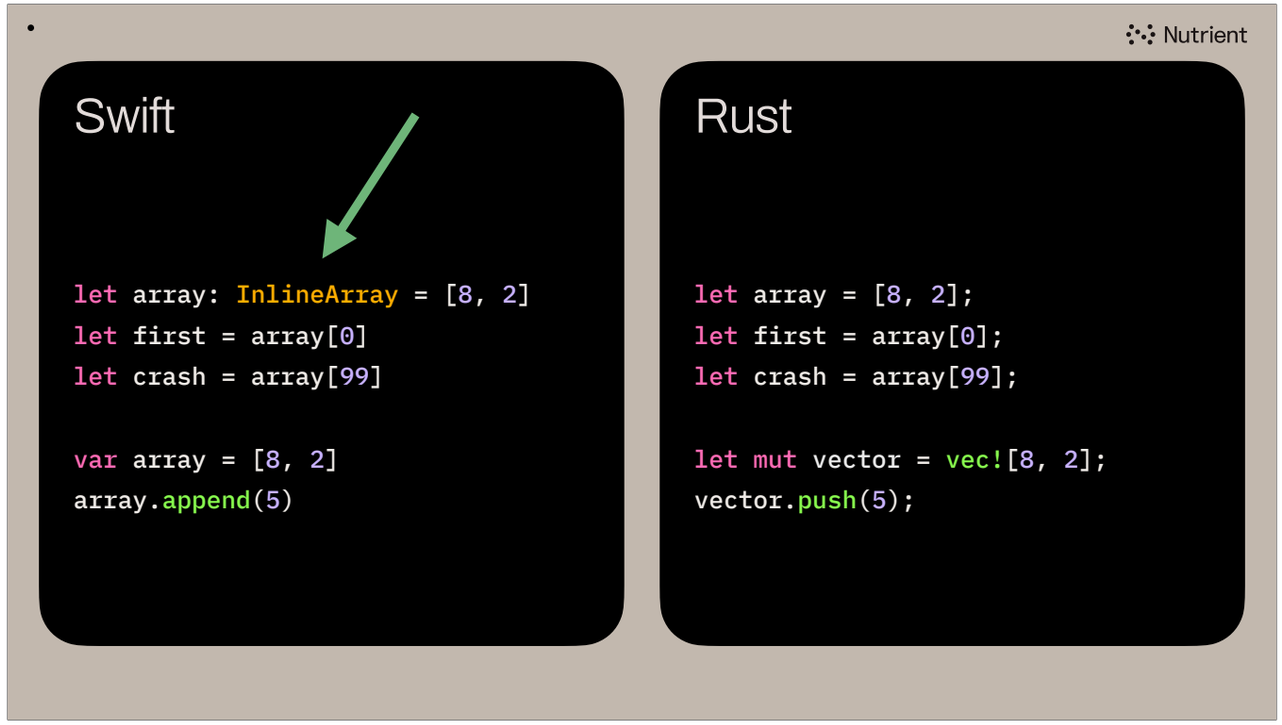
Let’s see what each language calls an array.



Swift is on the left. We know it will crash for out of bounds access on the third line. This looks similar to the Rust on the right. The only difference is the semicolons. Actually the Rust compiler flags the out of bounds access earlier as a build error, which is cool, but…

The more important difference is that in Rust arrays have inline storage with a fixed-size. This means they can be stack allocated, because the size of data on the stack must be known at compile time. This means you can't insert or remove objects from Rust arrays because the memory size must remain fixed.

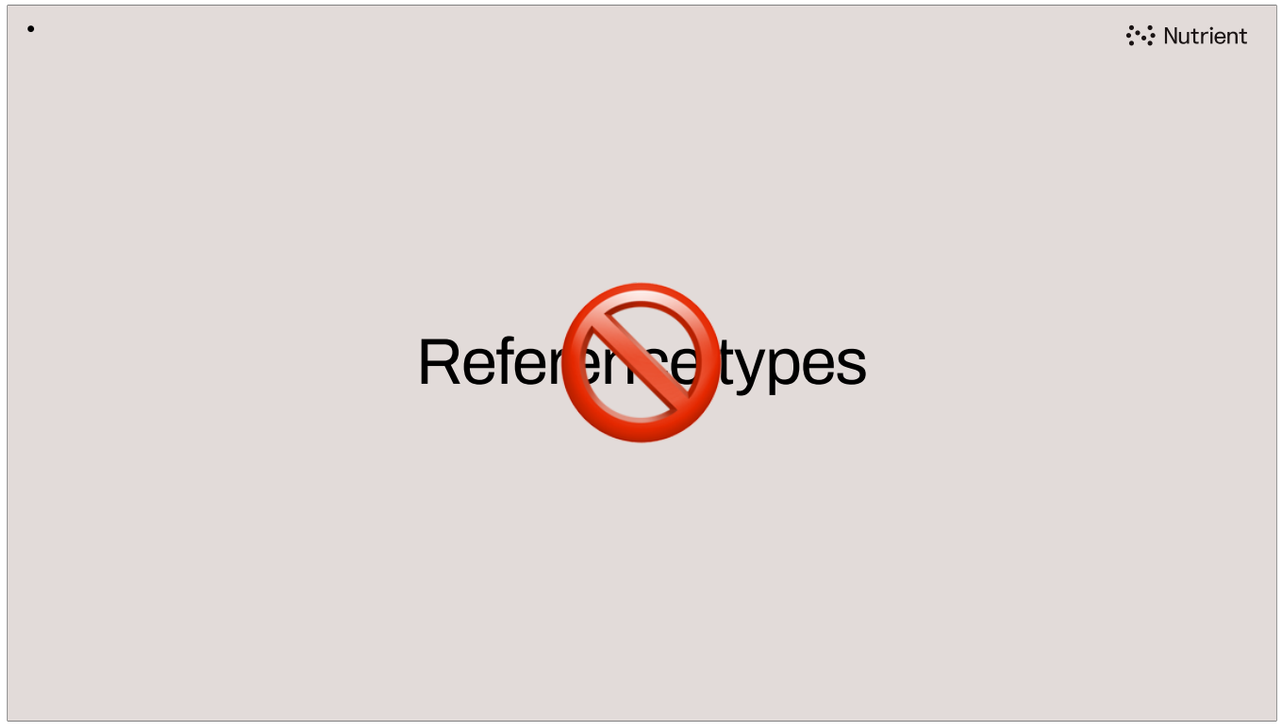
Instead, you use a different type called vector if you need to insert and remove. In Swift we usually just use the Array type and don't think about where the memory is allocated.



But actually since last year Swift has an `InlineArray` type, which uses inline storage and has the same constraints as a Rust array for when you need to really optimise performance.

Same capabilities, but things are introduced in a different order because the emphasis is different.

This comes back to progressive disclosure versus explicitness. Swift deliberately looks simpler at first, while Rust wants to be clearer about these details that can impact performance.



In Swift, we have classes and structs. With classes, you may need to know about more of the codebase to know why behaviours might happen, because of shared references and superclasses. Value types like structs are better for local reasoning.

I think most people would say that preferring structs is more Swifty. Rust goes full Swifty by not having classes or reference types as a language feature.

Nutrient

Value types only

- Rust has **no classes** or reference types
- Closest thing to subclassing is traits (like protocols)
- For reference counting, use wrapper types like `Arc<Mutex<T>>`

You can do object-oriented programming using a combination of structs, traits and reference counting provided by types, not the language itself.

Nutrient

Error handling

Swift has a lot of keywords for Error handling. It also has a Result type.

Nutrient

Error handling in Rust

```rust
fn fetch_speakers(year: i32) -> Result<Speakers, ConfError> {
    if year == 2025 {
        return Err(ConfError::Skipped);
    }

    let conference = Conference::new(year)?.speakers();

    Ok(conference)
}
```

Rust is more minimal: Result is how you handle errors. Leans on it’s expressive type system, which is pretty Swifty.

Nutrient

Error handling in Rust

```rust
fn fetch_speakers(year: i32) -> Result<Speakers, ConfError> {
    if year == 2025 {
        return Err(ConfError::Skipped);
    }

    let conference = Conference::new(year)?.speakers();

    Ok(conference)
}
```

The question mark operator looks like optional chaining in Swift, and can even work like that for a single line function.

But it’s more like `try` since it returns from the function on failure.

One of the more subtle syntaxes in Rust, but it results in very concise error propagation.

Nutrient

Memory management

Most languages in use these days use garbage collection, which is safe and doesn’t require thinking about memory too much, but it has a runtime processing cost. And memory use may be higher than it needs to be just before a sweep.

Swift and Rust are unusual together in how they handle memory by not using garbage collection.

Nutrient

Reference types in Swift

```swift
let conf = Conference(name: "iOSDevUK") // This is a class
let otherConf = conf // Cheap pointer copy + ARC
```

Let’s look at variables, which is the most fundamental way our programmes use memory.

In Swift, we have reference types.

When we assign one variable to another, we have two references to the same object, so modifying one modifies the other.

Assignment just copies a 64-bit pointer. Cheap.
The compiler also insert ARC retain and release calls.

Nutrient

Value types in Swift

```
var count = 5
var otherCount = count // Cheap value copy
```

With value types assignment copies the value so the two variables are independent.

Here we have an integer, which is a small value type, probably 32- or 64-bits, so this is cheap to copy again.

Nutrient

Value types in Swift

```
var string = "Really big string"
var otherString = string
string.append("more text") // Possibly expensive copy
```

For large values types, like a string needing 1MB of memory, assignment also gives us two independent variables.

Copying 1MB of memory sounds expensive, but behind the scenes Swift implements copy-on-write for strings and some other key data types. This means if we don’t modify either string they share the same memory.

The compiler makes sure this is safe. Yes, this sounds a lot like reference counting but with value types.

However if one string is modified, that 1MB of memory will be duplicated. This can give you a potentially expensive copy at a time that’s hard to predict.

Nutrient

Noncopyable types in Swift

```
struct NoncopyableString: ~Copyable {
    let value: String
}
```

But there’s more than just reference and value types.

Swift supports noncopyable reference types. This is a newer and less commonly used feature of Swift.

We mark a type as noncopyable using a sort of ‘negative protocol conformance’.

This NoncopyableString isn’t intended to be useful. It’s just an example.

Nutrient

Noncopyable types in Swift

```
struct NoncopyableString: ~Copyable {
    let value: String
}

let noncopyableString = NoncopyableString(value: "Really big string")
let other = noncopyableString

print("String is \(noncopyableString.value).")
```

❌ used after consume

What happens if like before we create one, assign it to a different variable? It’s a value type, so we can’t just make a second reference, and we can’t copy the value.

To see what happens, we need to use the first variable after the assignment.

We can’t do this because it wouldn’t be safe to have a single memory allocation shared in this way. You’ll see a compiler error saying the variable was used after consume.

We say that the value has *moved* into the other variable. The ownership has changed between the variables. The original variable has been consumed.

Nutrient

Noncopyable parameters in Swift

```
struct NoncopyableString: ~Copyable {
    let value: String
}

func doSomething(with noncopyableString: NoncopyableString) {
}
```

❌

parameter of noncopyable type must specify ownership

(consuming, borrowing, or inout)

Let’s look at passing noncopyable types into functions.

We already have a compiler error. We need to specify ownership, and it gives us three options: consuming, borrowing or inout.

Nutrient

Noncopyable parameters in Swift

```
struct NoncopyableString: ~Copyable {
    let value: String
}

func doSomething(with noncopyableString: consuming NoncopyableString) {
}

let noncopyableString = NoncopyableString(value: "Really big string")
doSomething(with: nonCopyableString)
print("String is \(noncopyableString.value).")
```

✗ used after consume

If we add consuming, then we get a situation like with the variable assignment.

The variable can no longer be used by the caller after passing it into the function.

The upside is the function owns the value so can do what it wants with the value, including escaping it from the function like assigning it to longer-lived memory like a property.

Nutrient

Noncopyable parameters in Swift

```
struct NoncopyableString: ~Copyable {
    let value: String
}

func doSomething(with noncopyableString: borrowing NoncopyableString) {
}

let noncopyableString = NoncopyableString(value: "Really big string")
doSomething(with: nonCopyableString)
print("String is \(noncopyableString.value).")
```

By changing the parameter from consuming to borrowing, the function can use the value with some restrictions because it doesn’t have ownership.

Callers are now able to continue using that value, but the value is no longer allowed to escape from the function.

We can’t modify a borrowed value.

Nutrient

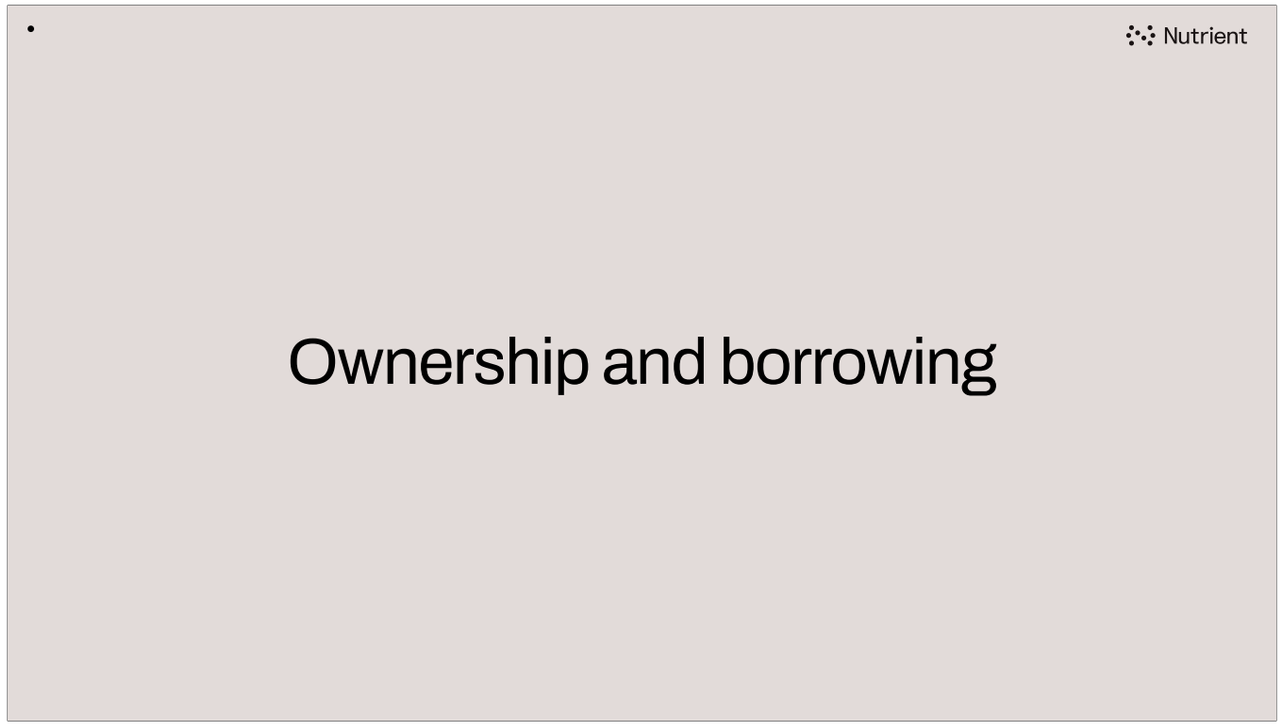
Noncopyable parameters in Swift

```
struct NoncopyableString: ~Copyable {
    let value: String
}

func doSomething(with noncopyableString: inout NoncopyableString) {
}

let noncopyableString = NoncopyableString(value: "Really big string")
doSomething(with: nonCopyableString)
print("String is \(noncopyableString.value).")
```

To modify the value in the function without taking ownership, we need the `inout` parameter modifier to create a mutable borrow.



We’ve just seen the basics of ownership and borrowing in Swift. We need to think about ownership specifically because the values can’t be copied.

The same concepts apply in Rust.

The difference is that in Rust the default is flipped: by default, types are noncopyable, so assignments move values.

Certain types like integers implement the `Copy` trait since their memory can be copied cheaply. They have implicit copying like in Swift.

Nutrient

Ownership in Rust

```
let string = String::from("Really big string");
let other_string = string;

println!("The string is {string}.");
```

❌ borrowed after move

For other types, notably `String`, are not implicitly copyable.

Same situation as we saw in Swift. You need to either follow the ownership and borrowing rules to safely work with a single memory allocation…

Nutrient

Ownership in Rust

```
let string = String::from("Really big string");
let other_string = string.clone();

println!("The string is {string}.");
```

…or you need to explicitly call `clone()` when you really need to duplicate the memory.

Nutrient

Ownership in Rust

```rust
fn debug_print(conference: Conference) {
    println!("The conference is {conference:?}.");
}
```

The syntax for ownership and borrowing is more concise in Rust.

Owning parameters are the default.

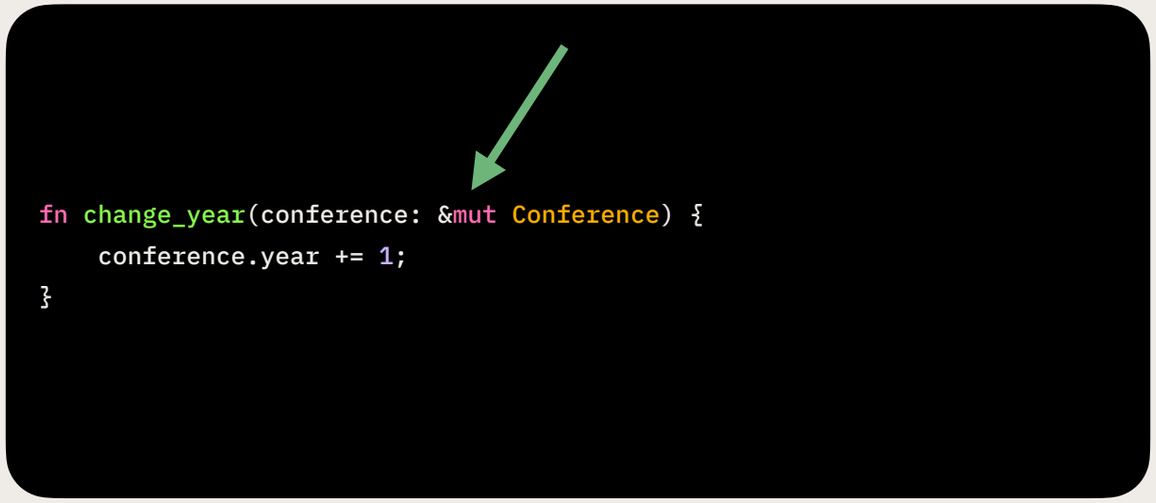
Nutrient

Borrowing in Rust

A borrowing reference uses an ampersand.

Nutrient

Mutable borrowing in Rust



Mutable borrowing uses ampersand mut.

Nutrient

Ownership and borrowing

Ownership is going to take a bit longer to get your head around. It’s a hard aspect of Rust, but I think it’s manageable. Compiler and AI will help you.

Nutrient

Closing

- Rust and Swift have a lot of similarities
  - Swift is sometimes Rusty
  - Rust is sometimes more Swifty than Swift
- Next steps:
  - Rust book
  - Build slowly with AI

• Rust and Swift developed around the same time with similar goals, so turned out with many similarities.
• If you like Swift, you'll probably like Rust too and won't find it that hard to learn.
• Newer Swift features going in a Rusty direction: InlineArray, macros, and noncopyable types.
• Rust is sometimes pursuing ideas we might call Swifty more aggressively than Swift. Pushes value types by not having reference types as a language feature. Leans more on the powerful type system.
• Learning languages shows you other design choices that can improve your understanding and skills with languages you already know.
• If you want to try Rust, the official Rust book is pretty good and a good place to start. Skim chapters because you'll know the same concepts from Swift or other languages.
• Building stuff with AI is a great way to learn things if you take the time to be heavily involved and understand what's going on.