Types

Mixing types

Some languages, like C, Perl and Python, like to be “helpful” to the programmer

Types

Mixing types

Some languages, like C, Perl and Python, like to be “helpful” to the programmer

Given code like `1.0 + 2` the compiler says “well, the programmer obviously meant `2` to be a floating point value, so I’ll change it to `2.0` for them”

Types

Mixing types

Some languages, like C, Perl and Python, like to be “helpful” to the programmer

Given code like `1.0 + 2` the compiler says “well, the programmer obviously meant `2` to be a floating point value, so I’ll change it to `2.0` for them”

The problem being that the compiler has to guess what the programmer “meant” and can’t always get that guess correct

Types

Mixing types

Some languages, like C, Perl and Python, like to be “helpful” to the programmer

Given code like `1.0 + 2` the compiler says “well, the programmer obviously meant `2` to be a floating point value, so I’ll change it to `2.0` for them”

The problem being that the compiler has to guess what the programmer “meant” and can’t always get that guess correct

So sometimes doing things that the programmer doesn’t expect, particularly in more complicated examples

Types

Mixing types

On the other hand, some languages, like Rust, try to be precise

Types

Mixing types

On the other hand, some languages, like Rust, try to be precise

The compiler says, “something wrong here, integers and floating points are very different things — the programmer must be confused, so I won’t compile this until they’ve fixed the problem”

Types

Mixing types

On the other hand, some languages, like Rust, try to be precise

The compiler says, “something wrong here, integers and floating points are very different things — the programmer must be confused, so I won’t compile this until they’ve fixed the problem”

A trade-off between languages that are “fast” or “easy” to program in, but the code might not be correct; against languages that need more time to program in, but the code is more likely to be correct

Aside

Coerce vs. Cast

Some terminology: “cast” is an explicit change of type, such as

```
double ns = (double)secs/1e9;
```

where `secs` is `int`

Aside

Coerce vs. Cast

Some terminology: “cast” is an explicit change of type, such as

```
double ns = (double)secs/1e9;
```

where `secs` is `int`

While “coerce” is an implicit change of type, such as

Aside

Coerce vs. Cast

Some terminology: “cast” is an explicit change of type, such as

```
double ns = (double)secs/1e9;
```

where `secs` is `int`

While “coerce” is an implicit change of type, such as

```
double x = 1/2;
```

Aside

Coerce vs. Cast

Some terminology: “cast” is an explicit change of type, such as

```
double ns = (double)secs/1e9;
```

where `secs` is `int`

While “coerce” is an implicit change of type, such as

```
double x = 1/2;
```

where the `int` value of the expression on the right is automatically changed to a `double` to store in `x`

Aside

Coerce vs. Cast

Some terminology: “cast” is an explicit change of type, such as

```
double ns = (double)secs/1e9;
```

where `secs` is `int`

While “coerce” is an implicit change of type, such as

```
double x = 1/2;
```

where the `int` value of the expression on the right is automatically changed to a `double` to store in `x`

It is usually better for code to be explicit, so many people (and languages) don’t like implicit casting

Aside

Coerce vs. Cast

Some terminology: “cast” is an explicit change of type, such as

```
double ns = (double)secs/1e9;
```

where `secs` is `int`

While “coerce” is an implicit change of type, such as

```
double x = 1/2;
```

where the `int` value of the expression on the right is automatically changed to a `double` to store in `x`

It is usually better for code to be explicit, so many people (and languages) don’t like implicit casting

The value of `x` above is 0.000

Types

Mixing types

A strongly typed language would refuse to compile such code, and require the programmer to get the types right, or insert explicit casts

Types

Mixing types

A strongly typed language would refuse to compile such code, and require the programmer to get the types right, or insert explicit casts

This is bad C code:

```
double x = 1;
int n = sqrt(2);
```

Types

Mixing types

A strongly typed language would refuse to compile such code, and require the programmer to get the types right, or insert explicit casts

This is bad C code:

```
double x = 1;
int n = sqrt(2);
```

This is better:

```
double x = 1.0;
int n = (int)sqrt(2.0);
```

as it makes the programmer's thinking much more clear

Types

Mixing types

Types are there to help you avoid bugs; don’t throw away this help!

Types

Mixing types

Types are there to help you avoid bugs; don't throw away this help!

Advanced exercise Read about the various kinds of casting in C++, such as `reinterpret_cast` and `static_cast`

Exercise What does Java do for `float x = 1.0;`? What is the correct code?

Types

Untyped: assembly language, BCPL, Forth, . . .

Types

Untyped: assembly language, BCPL, Forth, . . .

- up to the programmer how to interpret a value

Types

Untyped: assembly language, BCPL, Forth, . . .

- up to the programmer how to interpret a value
- all values are just presented as a machine byte or word

Types

Untyped: assembly language, BCPL, Forth, ...

- up to the programmer how to interpret a value
- all values are just presented as a machine byte or word

Not so widely used as these days for general computing as types are seen as an essential aid to the programmer

Types

Untyped: assembly language, BCPL, Forth, ...

- up to the programmer how to interpret a value
- all values are just presented as a machine byte or word

Not so widely used as these days for general computing as types are seen as an essential aid to the programmer

Though assembly language is still more widely used than you might expect

Types

Untyped: assembly language, BCPL, Forth, ...

- up to the programmer how to interpret a value
- all values are just presented as a machine byte or word

Not so widely used as these days for general computing as types are seen as an essential aid to the programmer

Though assembly language is still more widely used than you might expect

And even assembly languages do tend to treat floating point numbers differently from integers/pointers/bit patterns

Types

Feet

- BCPL: You shoot yourself somewhere in the leg—you can't get any finer resolution than that

Types

Feet

- BCPL: You shoot yourself somewhere in the leg—you can't get any finer resolution than that
- Forth: Foot yourself in the shoot

Types

Comparing these kinds of types:

- Dynamic: flexibility for the programmer, particularly in prototyping where fast coding through few restrictions is important

Types

Comparing these kinds of types:

- Dynamic: flexibility for the programmer, particularly in prototyping where fast coding through few restrictions is important
- Static: types checked at compile time, catching some bugs in the source before the program is run. Consequently, compilation is usually slower, but the result is likely less buggy

Types

Comparing these kinds of types:

- Dynamic: flexibility for the programmer, particularly in prototyping where fast coding through few restrictions is important
- Static: types checked at compile time, catching some bugs in the source before the program is run. Consequently, compilation is usually slower, but the result is likely less buggy
- Untyped: no type errors possible, and no checking done for the programmer

Types

There are other differences, too

Types

There are other differences, too

We can look at what each do when presented with source code like `a+b`

Types

There are other differences, too

We can look at what each do when presented with source code like `a+b`

- what a compiler needs to do with it

Types

There are other differences, too

We can look at what each do when presented with source code like `a+b`

- what a compiler needs to do with it
- what happens when the code is running

Types

There are other differences, too

We can look at what each do when presented with source code like `a+b`

- what a compiler needs to do with it
- what happens when the code is running

An interpreter would need to do both stages above while executing

Types

For any such operation, a compiler for a dynamic language will need to generate and output code that

- checks if `a` is a number
- checks if `b` is a number
- if so call the appropriate add function
- else does some coercions then adds; or just signals an error, as appropriate

Types

Then at runtime all this rather complicated code will be executed

Types

Then at runtime all this rather complicated code will be executed

There must be a runtime check (in the absence of clever optimisations) since the values of `a` and `b` can be of any types

Types

Then at runtime all this rather complicated code will be executed

There must be a runtime check (in the absence of clever optimisations) since the values of `a` and `b` can be of any types

Thus a lot of checking overhead before actually doing the expected operation: this can easily be much larger than the operation itself!

Types

Then at runtime all this rather complicated code will be executed

There must be a runtime check (in the absence of clever optimisations) since the values of `a` and `b` can be of any types

Thus a lot of checking overhead before actually doing the expected operation: this can easily be much larger than the operation itself!

This can be many dozens of CPU instructions overhead

Types

Then at runtime all this rather complicated code will be executed

There must be a runtime check (in the absence of clever optimisations) since the values of `a` and `b` can be of any types

Thus a lot of checking overhead before actually doing the expected operation: this can easily be much larger than the operation itself!

This can be many dozens of CPU instructions overhead

Exercise Read the ECMA (JavaScript) standard to discover the 10 step process that it requires for addition

Types

Then at runtime all this rather complicated code will be executed

There must be a runtime check (in the absence of clever optimisations) since the values of `a` and `b` can be of any types

Thus a lot of checking overhead before actually doing the expected operation: this can easily be much larger than the operation itself!

This can be many dozens of CPU instructions overhead

Exercise Read the ECMA (JavaScript) standard to discover the 10 step process that it requires for addition

Exercise Investigate how an add operation gets executed in Python

Types

Static. The compiler will determine the types of `a` and `b` (and therefore the types of the values stored in `a` and `b`) and generate

- code for the appropriate add operation (maybe with appropriate coercions, if the language allows such things)

Types

Static. The compiler will determine the types of `a` and `b` (and therefore the types of the values stored in `a` and `b`) and generate

- code for the appropriate add operation (maybe with appropriate coercions, if the language allows such things)

It does not need to generate code to check the values of `a` and `b` as they *must* be of the correct types when the code reaches that point

Types

Static. The compiler will determine the types of `a` and `b` (and therefore the types of the values stored in `a` and `b`) and generate

- code for the appropriate add operation (maybe with appropriate coercions, if the language allows such things)

It does not need to generate code to check the values of `a` and `b` as they *must* be of the correct types when the code reaches that point

At runtime just this simple add operation will be executed

Types

Static. The compiler will determine the types of `a` and `b` (and therefore the types of the values stored in `a` and `b`) and generate

- code for the appropriate add operation (maybe with appropriate coercions, if the language allows such things)

It does not need to generate code to check the values of `a` and `b` as they *must* be of the correct types when the code reaches that point

At runtime just this simple add operation will be executed

This might be just a single CPU instruction

Types

Static. The compiler will determine the types of `a` and `b` (and therefore the types of the values stored in `a` and `b`) and generate

- code for the appropriate add operation (maybe with appropriate coercions, if the language allows such things)

It does not need to generate code to check the values of `a` and `b` as they *must* be of the correct types when the code reaches that point

At runtime just this simple add operation will be executed

This might be just a single CPU instruction

No runtime checks are needed

Types

Untyped. The compiler will output code to add the values (presumably an integer add) regardless of what the programmer thinks they happen to be

Types

Untyped. The compiler will output code to add the values (presumably an integer add) regardless of what the programmer thinks they happen to be

At runtime this simple operation will be executed

Types

Untyped. The compiler will output code to add the values (presumably an integer add) regardless of what the programmer thinks they happen to be

At runtime this simple operation will be executed

There's nothing that it could check!

Types

Untyped. The compiler will output code to add the values (presumably an integer add) regardless of what the programmer thinks they happen to be

At runtime this simple operation will be executed

There's nothing that it could check!

Exercise Compare the code output by a static language and an untyped language

Types

If I have a drawer marked “Socks” I don’t need to check what comes out of it before I put them on my feet

Types

If I have a drawer marked “Socks” I don’t need to check what comes out of it before I put them on my feet

If I have an unmarked drawer, I need to look at what I get, first

Types

A brief peek into the Object Oriented world. . .

Types

A brief peek into the Object Oriented world. . .

It's not just variables or their values that can be static or dynamic: in the case of OO method lookup we can also see significant differences

Types

A brief peek into the Object Oriented world. . .

It's not just variables or their values that can be static or dynamic: in the case of OO method lookup we can also see significant differences

Suppose we have code `a.foo()`

Types

Dynamic lookup. The compiler will generate

- code to choose the correct method to call on the current value of `a` (determined by the type of the current value)
- then code to call the chosen method

Types

Dynamic lookup. The compiler will generate

- code to choose the correct method to call on the current value of `a` (determined by the type of the current value)
- then code to call the chosen method

At runtime the code to choose the method needs to run first: the choosing is done at runtime

Types

Dynamic lookup. The compiler will generate

- code to choose the correct method to call on the current value of `a` (determined by the type of the current value)
- then code to call the chosen method

At runtime the code to choose the method needs to run first: the choosing is done at runtime

Again, overhead before the method itself can be run

Types

Dynamic lookup. The compiler will generate

- code to choose the correct method to call on the current value of `a` (determined by the type of the current value)
- then code to call the chosen method

At runtime the code to choose the method needs to run first: the choosing is done at runtime

Again, overhead before the method itself can be run

In a dynamic lookup, calling a method can be considerably slower to execute than calling a function

Types

Static lookup. The compiler will determine the type of `a`, find the appropriate method, and output

- code to call the method

Types

Static lookup. The compiler will determine the type of `a`, find the appropriate method, and output

- code to call the method

At runtime the code of the method is called directly on the value of `a` as the lookup has already been done by the compiler

Types

Static lookup. The compiler will determine the type of `a`, find the appropriate method, and output

- code to call the method

At runtime the code of the method is called directly on the value of `a` as the lookup has already been done by the compiler

So for a static lookup calling a method is just as fast to execute as calling a function

Types

Static lookup. The compiler will determine the type of `a`, find the appropriate method, and output

- code to call the method

At runtime the code of the method is called directly on the value of `a` as the lookup has already been done by the compiler

So for a static lookup calling a method is just as fast to execute as calling a function

Though this is not the whole story: more on this later

Types

Untyped. No OO possible!

Types

Thus we have a tradeoff for static vs. dynamic types. We get either:

- slower compiler, more compile-time checking, faster running code; against
- faster compiler, slower running code

Types

Thus we have a tradeoff for static vs. dynamic types. We get either:

- slower compiler, more compile-time checking, faster running code; against
- faster compiler, slower running code

Remember: for some people, a fast compile-run-debug cycle is more important than fast (or correct) code!

Types

So, it seems, static is faster to execute and is therefore “better”

Types

So, it seems, static is faster to execute and is therefore “better”

But the hidden point in dynamic is “the current value of `a`”

Types

So, it seems, static is faster to execute and is therefore “better”

But the hidden point in dynamic is “the current value of `a`”

In many OO languages the type of the object held in variable `a` can vary at runtime, so the appropriate method can vary at runtime

Aside

Note that the phrase “the type of the object held in a variable can vary” is ambiguous

Aside

Note that the phrase “the type of the object held in a variable can vary” is ambiguous

By this we might mean something like `x = 1` at one point and `x = "cat"` at another

Aside

Note that the phrase “the type of the object held in a variable can vary” is ambiguous

By this we might mean something like `x = 1` at one point and `x = "cat"` at another

So the value contained in `x` changes, and they have different types

Aside

Note that the phrase “the type of the object held in a variable can vary” is ambiguous

By this we might mean something like `x = 1` at one point and `x = "cat"` at another

So the value contained in `x` changes, and they have different types

Conversely, there are languages that allow you to change to type of the value itself: the value stays the same, but its type is allowed to change

Aside

Note that the phrase “the type of the object held in a variable can vary” is ambiguous

By this we might mean something like `x = 1` at one point and `x = "cat"` at another

So the value contained in `x` changes, and they have different types

Conversely, there are languages that allow you to change to type of the value itself: the value stays the same, but its type is allowed to change

Exercise Think about this in the context of type hierarchies, e.g., casting an instance of `Dog` to an instance of `Animal`

Types

More commonly, we could have:

```
Animal a = new Animal(...)
...
a = new Dog(...)
```

when `Dog` is a subclass of `Animal`, we have `a` containing values of different types

Types

In any case, the same line of code `a.foo()` might need a different method each time you come to it as the type of the value of `a` is potentially different each time

Types

In any case, the same line of code `a.foo()` might need a different method each time you come to it as the type of the value of `a` is potentially different each time

Even in static languages: see later

Types

Furthermore, some languages even allow you to create and add new methods dynamically as the program is running, so even if the type of the value of `a` is unchanged, the correct method to call still might have changed!

Types

Furthermore, some languages even allow you to create and add new methods dynamically as the program is running, so even if the type of the value of `a` is unchanged, the correct method to call still might have changed!

This is the essence of the flexibility of dynamic languages

Types

Furthermore, some languages even allow you to create and add new methods dynamically as the program is running, so even if the type of the value of `a` is unchanged, the correct method to call still might have changed!

This is the essence of the flexibility of dynamic languages

The cost is the execution speed

Types

Furthermore, some languages even allow you to create and add new methods dynamically as the program is running, so even if the type of the value of `a` is unchanged, the correct method to call still might have changed!

This is the essence of the flexibility of dynamic languages

The cost is the execution speed

And the ability of the programmer to understand what is happening

Types

```
for i in range(10):
    if random.random() > 0.5:
        x = "hello"
    else:
        x = 42
    print(x + x)
```

Again, it is arguably bad style to do this without good reason

Types

Duck typing is a highly dynamic kind of typing: examples are Python, JavaScript, Common Lisp, Ruby

Types

Duck typing is a highly dynamic kind of typing: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the runtime examines the current value of `a` to see if there is a `foo` method currently defined on it and calls it if it find one

Types

Duck typing is a highly dynamic kind of typing: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the runtime examines the current value of `a` to see if there is a `foo` method currently defined on it and calls it if it find one

It's not worried about the class of `a` (such a language might not even have classes), only whether an appropriate method named `foo` exists at that point in time

Types

Duck typing is a highly dynamic kind of typing: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the runtime examines the current value of `a` to see if there is a `foo` method currently defined on it and calls it if it find one

It's not worried about the class of `a` (such a language might not even have classes), only whether an appropriate method named `foo` exists at that point in time

It is a runtime error if no method is found

Types

Duck typing is a highly dynamic kind of typing: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the runtime examines the current value of `a` to see if there is a `foo` method currently defined on it and calls it if it find one

It's not worried about the class of `a` (such a language might not even have classes), only whether an appropriate method named `foo` exists at that point in time

It is a runtime error if no method is found

The same line of code may or may not work depending on the current value of `a`!

Types

Duck typing is a highly dynamic kind of typing: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the runtime examines the current value of `a` to see if there is a `foo` method currently defined on it and calls it if it find one

It's not worried about the class of `a` (such a language might not even have classes), only whether an appropriate method named `foo` exists at that point in time

It is a runtime error if no method is found

The same line of code may or may not work depending on the current value of `a`!

"If it walks like a duck and talks like a duck, then it is a duck"

Types

Exercise Consider the Python

```
def two10(n):
    for i in range(10):
        n = 2*n
    return n

two10(1)
two10(1.0)
two10("1")
two10(two10)
```

Types

Next: how types are expressed in languages

Types

Next: how types are expressed in languages

In some languages you don’t need to declare your variables, e.g., Python

Types

Next: how types are expressed in languages

In some languages you don't need to declare your variables, e.g., Python

This is generally regarded as a bad feature as it leaves open the easy bug of misspelling variables, e.g., `complier` and `compiler`

Types

Next: how types are expressed in languages

In some languages you don't need to declare your variables, e.g., Python

This is generally regarded as a bad feature as it leaves open the easy bug of misspelling variables, e.g., `complier` and `compiler`

Though some languages, e.g., Python, do do some runtime checking to mitigate this

Types

While declaring variables, you may or may not need to declare types

Types

While declaring variables, you may or may not need to declare types

In dynamically typed languages you (generally) don't declare the type of variables, e.g., `var x = 7;` in JavaScript

Types

While declaring variables, you may or may not need to declare types

In dynamically typed languages you (generally) don't declare the type of variables, e.g., `var x = 7;` in JavaScript

It doesn't really make sense to declare the type of a variable in a dynamic language as it's the value that has the type, not the variable

Types

On the other hand in a statically typed language the type of the variable is important

Types

On the other hand in a statically typed language the type of the variable is important

And they can have a few different ways to designate types of variables in the source code

Types

Manifest Typing: where the program code includes the types of variables, e.g., C

```
int inc(int n)
{
  return n+1;
}
```

Types

Implicit Typing: where the compiler infers any types it needs (as much as it can), e.g., a Haskell function definition

```
inc x = x + 1
```

and Haskell determines the type of `inc` to be `Num a => a -> a`

Types

Or both, as in Rust:

```
fn fix(x: f64) -> i32 { ... }
...
let y = fix(z);
```

and Rust determines the type of `y` to be `i32` and `z` to be `f64`

Types

Implicitly typed languages allow (or require, in ambiguous code) the programmer to include type annotations if they want

Types

Implicitly typed languages allow (or require, in ambiguous code) the programmer to include type annotations if they want

`let x: f64 = 42.0`	explicit
`let x = 42.0`	inferred

Types

Implicitly typed languages allow (or require, in ambiguous code) the programmer to include type annotations if they want

`let x: f64 = 42.0`	explicit
`let x = 42.0`	inferred

And Go:

`var i int = 23`	explicit
`i := 23`	inferred

Types

Implicitly typed languages allow (or require, in ambiguous code) the programmer to include type annotations if they want

```
let x: f64 = 42.0
let x = 42.0
```

explicit
inferred

And Go:

```
var i int = 23
i := 23
```

explicit
inferred

And Java:

```
int i = 23;
var i = 23;
```

explicit
inferred

Types

As a middle path, some languages, in particular Rust, allow *most* type annotations to be implicit while requiring others to be manifest

Types

As a middle path, some languages, in particular Rust, allow *most* type annotations to be implicit while requiring others to be manifest

Rust requires explicit type annotations on function declarations

Types

As a middle path, some languages, in particular Rust, allow *most* type annotations to be implicit while requiring others to be manifest

Rust requires explicit type annotations on function declarations

This is a language design choice and the justification is that it makes function APIs explicit and thus combining code from several places much less error prone

Types

As a middle path, some languages, in particular Rust, allow *most* type annotations to be implicit while requiring others to be manifest

Rust requires explicit type annotations on function declarations

This is a language design choice and the justification is that it makes function APIs explicit and thus combining code from several places much less error prone

```
fn triple(n: i32) -> i32 ...
```

Types

As a middle path, some languages, in particular Rust, allow *most* type annotations to be implicit while requiring others to be manifest

Rust requires explicit type annotations on function declarations

This is a language design choice and the justification is that it makes function APIs explicit and thus combining code from several places much less error prone

```
fn triple(n: i32) -> i32 ...
```

When want `triple` you know exactly how to use it, quite the opposite to duck typing

Types

Compare with

```
fn triple(n: i32) -> (i32,i32,i32) ...
```

Types

Compare with

```
fn triple(n: i32) -> (i32,i32,i32) ...
```

Without the types the programmer is less sure on what is happening!

Types

Aside

Mathematically speaking, a function includes its domain and codomain so sqr : $[0..\infty) \to [0..\infty)$ is different to
sqr : $(-\infty..\infty) \to [0..\infty)$ is different to sqr : $[0..\infty) \to (-\infty..\infty)$

Types

Aside

Mathematically speaking, a function includes its domain and codomain so sqr : $[0..\infty) \to [0..\infty)$ is different to
sqr : $(-\infty..\infty) \to [0..\infty)$ is different to sqr : $[0..\infty) \to (-\infty..\infty)$

In programming terms: `uint -> uint` vs `int -> uint` vs `uint -> int`

Types

Aside

Mathematically speaking, a function includes its domain and codomain so sqr : $[0..\infty) \to [0..\infty)$ is different to
sqr : $(-\infty..\infty) \to [0..\infty)$ is different to sqr : $[0..\infty) \to (-\infty..\infty)$

In programming terms: `uint -> uint` vs `int -> uint` vs `uint -> int`

And what about other sizes of int, or doubles or whatever?

Types

Aside

Mathematically speaking, a function includes its domain and codomain so sqr : $[0..\infty) \to [0..\infty)$ is different to sqr : $(-\infty..\infty) \to [0..\infty)$ is different to sqr : $[0..\infty) \to (-\infty..\infty)$

In programming terms: `uint -> uint` vs `int -> uint` vs `uint -> int`

And what about other sizes of int, or doubles or whatever?

This is important as documentation of the function, so helps use: if this is compiled into a library and I want to use it, how should I use it?

Types

Aside

Explicit: more work for the programmer; code is clear on types and better documented

Types

Aside

Explicit: more work for the programmer; code is clear on types and better documented

Implicit: less work for the programmer; code can be harder for the programmer to understand

Types

Aside

Explicit: more work for the programmer; code is clear on types and better documented

Implicit: less work for the programmer; code can be harder for the programmer to understand

Or easier when the types get complicated

Types

Aside

Explicit: more work for the programmer; code is clear on types and better documented

Implicit: less work for the programmer; code can be harder for the programmer to understand

Or easier when the types get complicated

In

```
var stream = Files.newInputStream(path);
```

the programmer doesn't have be bothered about what the type of `stream` should be

Types

Exercise And more mixed. For example, Rust allows mixed explicit and implicit in single expressions, e.g.,

```
let vals: Vec<_> = something
```

tells the compiler that `something` returns a vector of things, but lets the compiler infer what type the things are. Read about why this is done