Types

So:

Types

So:

- Manifest: allows for a simpler compiler as it doesn't have to work so hard. Requires the programmer to think explicitly about the types

Types

So:

- Manifest: allows for a simpler compiler as it doesn't have to work so hard. Requires the programmer to think explicitly about the types
- Mixed: the programmer thinks about types, but a lot of the hard work is done by the compiler. The code is moderately explicit about types

Types

So:

- Manifest: allows for a simpler compiler as it doesn't have to work so hard. Requires the programmer to think explicitly about the types
- Mixed: the programmer thinks about types, but a lot of the hard work is done by the compiler. The code is moderately explicit about types
- Implicit: allows for simpler code, but requires a much more complex compiler to do the type inference. Possibly the code is harder to understand by the programmer (less "documentation" in code)

Types

So:

- Manifest: allows for a simpler compiler as it doesn't have to work so hard. Requires the programmer to think explicitly about the types
- Mixed: the programmer thinks about types, but a lot of the hard work is done by the compiler. The code is moderately explicit about types
- Implicit: allows for simpler code, but requires a much more complex compiler to do the type inference. Possibly the code is harder to understand by the programmer (less "documentation" in code)

A trade-off of compiler speed against coding speed

Types

Python allows optional type declarations for variables:

```
def double(n: int):
    return n+n
```

Types

Python allows optional type declarations for variables:

```
def double(n: int):
    return n+n
```

Though this is purely documentary and not checked by the runtime: `double(3.14) -> 6.28` and `double('ha') -> 'haha'`

Types

Such documentary type declarations (also called *type hints*)

- make the code easier to understand for the programmers
- makes the code easier to refactor
- helps IDEs (e.g., autocompletion)
- helps the programmer (not the compiler) catch bugs

Types

Such documentary type declarations (also called *type hints*)

- make the code easier to understand for the programmers
- makes the code easier to refactor
- helps IDEs (e.g., autocompletion)
- helps the programmer (not the compiler) catch bugs

Exercise There are several static type checkers for Python (Mypy, Pytype, Pyright, Pyre, . . .). Why do these exist?

Types

Exercise Read about *Hindley-Milner* type systems

Exercise Elements of type inference is being adopted by some traditional explicitly typed languages. Why? Read about `auto` in C++, `var` in C# and `var` in Java

Exercise Think about type inference in the presence of automatic type coercions (weak typing)

Types

Static types are often further divided:

- Monomorphic/Lexical: variables have a single, definite type, so you can type-check purely on the variables

  ```
  int f(int x) { ... y = x; ... }
  ```

  A very common approach

Types

Polymorphism

- Polymorphic: types can be shown by *type variables*, e.g.,

  ```
  push: a * [a] -> [a]
  ```

  or

  ```
  template <class T>
  List<T> push(T x, List<T> l)
  ```

Types

Polymorphism

Or

```
public static <T> List<T> push(T x, List<T> l)
```

or

```
func push[T](x T, l []T) []T
```

where `a` and `T` are variables that stand for *types*, not values

So `push` is a function that takes a value of some type, a list of values of the same type and returns a list of values of that type

Types

Polymorphism

The idea of polymorphism seems to originate as far back as 1967 (Christopher Strachey)

Types

Polymorphism

The idea of polymorphism seems to originate as far back as 1967 (Christopher Strachey)

Polymorphic ("many shaped") functions are notionally functions that work on many types

Types

Polymorphism

The idea of polymorphism seems to originate as far back as 1967 (Christopher Strachey)

Polymorphic ("many shaped") functions are notionally functions that work on many types

That is, you could give it an argument of type `A` or an argument of type `B` and the function would (a) work and (b) do the "same thing" to the argument regardless of the actual type of the argument, e.g., `push`, above

Types

Polymorphism

The idea of polymorphism seems to originate as far back as 1967 (Christopher Strachey)

Polymorphic (“many shaped”) functions are notionally functions that work on many types

That is, you could give it an argument of type `A` or an argument of type `B` and the function would (a) work and (b) do the “same thing” to the argument regardless of the actual type of the argument, e.g., `push`, above

The function `push` works the same on lists of integers and lists of strings

Types

Polymorphism

In other circumstances, if you defined a function to take a value of one type and you gave it a value of another type, that would be an error

Types

Polymorphism

In other circumstances, if you defined a function to take a value of one type and you gave it a value of another type, that would be an error

After all, this is one of the reasons types are used: to catch the cases where you use a value of the wrong type

Types

Polymorphism

Polymorphism is where a function (or method) can be called on more than one type, e.g., `push` doesn't care what it's a list of, as all lists are built the same way

Types

Polymorphism

Polymorphism is where a function (or method) can be called on more than one type, e.g., `push` doesn't care what it's a list of, as all lists are built the same way

Polymorphism is really about presenting a single API to the programmer and it works on more than one type

Types

Polymorphism

Polymorphism is where a function (or method) can be called on more than one type, e.g., `push` doesn't care what it's a list of, as all lists are built the same way

Polymorphism is really about presenting a single API to the programmer and it works on more than one type

But the word "polymorphic" has expanded to mean something more complicated

Types

Overloading

The concept of *overloading* has been around for a long time

Types

Overloading

The concept of *overloading* has been around for a long time

Some languages (e.g., Java, C++, but not C) allow:

Types
Overloading

The concept of *overloading* has been around for a long time

Some languages (e.g., Java, C++, but not C) allow:

```
int f(int x) { return -x; }
double f(double x) { return 2.0*x; }
```

Types

Overloading

The concept of *overloading* has been around for a long time

Some languages (e.g., Java, C++, but not C) allow:

```
int f(int x) { return -x; }
double f(double x) { return 2.0*x; }
```

Multiple different functions with the same name

Types
Overloading

The concept of *overloading* has been around for a long time

Some languages (e.g., Java, C++, but not C) allow:

```
int f(int x) { return -x; }
double f(double x) { return 2.0*x; }
```

Multiple different functions with the same name

The compiler can distinguish which `f` we mean by the argument types: `f(2)` means the `int` function; while `f(2.0)` means the `double` function

Types

Overloading

And for `f(x)` the compiler looks at the declared type for `x` to see which `f` to use

Types
Overloading

And for `f(x)` the compiler looks at the declared type for `x` to see which `f` to use

And *different* chunks of code are compiled for each function

Types
Overloading

Aside: another reason why we need to be careful to distinguish between, say, `2` and `2.0`

Aside: another reason why we need to be careful to distinguish between, say, `2` and `2.0`

Exercise Think about the call `f(3/2)`

Types

Overloading

The function bodies can be completely different: it's almost incidental that the functions have the same name

Types

Overloading

The function bodies can be completely different: it's almost incidental that the functions have the same name

Though it would be sensible programming to have all instances of `f` do the same kind of thing on their arguments

Types

Overloading

The function bodies can be completely different: it's almost incidental that the functions have the same name

Though it would be sensible programming to have all instances of `f` do the same kind of thing on their arguments

Overloading does not *prevent* you making the various `f`s do wildly different things: but doing this would only make understanding your code harder

Types

Overloading

So the type of the argument determines what happens:
`f(2)` is compiled as a call to the first
`f(2.0)` is compiled as a call to the second

So the type of the argument determines what happens:
`f(2)` is compiled as a call to the first
`f(2.0)` is compiled as a call to the second

In fact, in a typical implementation, the compiler internally renames ("name mangling") the two functions as (something like) `f_int` and `f_double`, so giving them distinct names

Types

Overloading

It then (in effect) rewrites your code and replaces `f` everywhere as appropriate

Types

Overloading

It then (in effect) rewrites your code and replaces `f` everywhere as appropriate

It writes “normal” functions

`int f_int(int x) { return -x; }` and
`double f_double(double x) { return 2.0*x; }`,

and then

`f(2)` is replaced `f_int(2)`
`f(2.0)` is replaced by `f_double(2.0)`

Types

Overloading

It then (in effect) rewrites your code and replaces `f` everywhere as appropriate

It writes "normal" functions

`int f_int(int x) { return -x; }` and
`double f_double(double x) { return 2.0*x; }`,

and then

`f(2)` is replaced `f_int(2)`
`f(2.0)` is replaced by `f_double(2.0)`

It then compiles this "rewritten" code

Types
Overloading

Overloading is very widespread and appears in a limited way in lots of languages: common functions like + are often overloaded

Aside on Operators

Remember that operators like + and $*$ are just convenient syntax for the expected underlying functions or methods and otherwise are not particularly special

Aside on Operators

Remember that operators like + and $*$ are just convenient syntax for the expected underlying functions or methods and otherwise are not particularly special

You can write `1 + 2` rather than having to write `add(1, 2)` or `(add 1 2)` or `(1).__add__(2)`

Aside on Operators

Remember that operators like + and `*` are just convenient syntax for the expected underlying functions or methods and otherwise are not particularly special

You can write `1 + 2` rather than having to write `add(1, 2)` or `(add 1 2)` or `(1).__add__(2)`

Many languages overload operators, so, for example, allowing `int+int` and `double+double` values, sometimes strings, too

Aside on Operators

Some languages allow mixed types, too: `int+double` and `double+int`, as in `1 + 2.3`

Aside on Operators

Some languages allow mixed types, too: `int+double` and `double+int`, as in `1 + 2.3`

These can all refer to different underlying functions. E.g., `int+double` would likely coerce its first argument to a `double` before doing a `double+double` add. This is different from what `double+int` needs to do

Aside on Operators

OCaml doesn't overload +, but uses + for integer addition and `+.` for float addition

Aside on Operators

OCaml doesn't overload +, but uses + for integer addition and `+.` for float addition

BCPL, being untyped, didn't support float arithmetic for a long time (as the hardware of the time didn't either!), but later added it with non-overloaded operators like #+ and #*

Aside on Operators

OCaml doesn't overload +, but uses + for integer addition and `+.` for float addition

BCPL, being untyped, didn't support float arithmetic for a long time (as the hardware of the time didn't either!), but later added it with non-overloaded operators like `#+` and `#*`

A strongly typed language might overload `int+int` and `double+double` but not `int+double` or `double+int`, disallowing implicit coercion

Aside on Operators

OCaml doesn't overload +, but uses + for integer addition and `+.` for float addition

BCPL, being untyped, didn't support float arithmetic for a long time (as the hardware of the time didn't either!), but later added it with non-overloaded operators like `#+` and `#*`

A strongly typed language might overload `int+int` and `double+double` but not `int+double` or `double+int`, disallowing implicit coercion

Exercise Some languages (e.g., C++, Rust, Python) allow you to define your own methods on operators, while others don't (e.g., Java). Investigate

Types

Overloading/Polymorphism

So overloading is a way of having *different* chunks of code use the same function name

Types

Overloading/Polymorphism

So overloading is a way of having *different* chunks of code use the same function name

The polymorphism we saw earlier is different from overloading

Types

Overloading/Polymorphism

So overloading is a way of having *different* chunks of code use the same function name

The polymorphism we saw earlier is different from overloading

E.g., `length` to return the length of a list

Types

Overloading/Polymorphism

So overloading is a way of having *different* chunks of code use the same function name

The polymorphism we saw earlier is different from overloading

E.g., `length` to return the length of a list

Here, the *same* function code works on many types of list

Types

Overloading/Polymorphism

So overloading is a way of having *different* chunks of code use the same function name

The polymorphism we saw earlier is different from overloading

E.g., `length` to return the length of a list

Here, the *same* function code works on many types of list

There is just one chunk of code that works on multiple types

Types

Overloading/Polymorphism

So overloading is a way of having *different* chunks of code use the same function name

The polymorphism we saw earlier is different from overloading

E.g., `length` to return the length of a list

Here, the *same* function code works on many types of list

There is just one chunk of code that works on multiple types

`length [2]` (list of integers) runs the *same code* as
`length ["hello" "world"]` (list of strings)

Types

Overloading/Polymorphism

So overloading is a way of having *different* chunks of code use the same function name

The polymorphism we saw earlier is different from overloading

E.g., `length` to return the length of a list

Here, the *same* function code works on many types of list

There is just one chunk of code that works on multiple types

`length [2]` (list of integers) runs the *same code* as
`length ["hello" "world"]` (list of strings)

`length` doesn't care about the types of its arguments

Types

Overloading/Polymorphism

Beware of overloading disguised as polymorphism:

```
template <class T>        // T is a type variable
T f(T x) { return -x; }
```

in C++ defining a function `f` taking a value of type `T` and returning a value of type `T`, for all types `T`. Similarly:

```
// T any type that implements negation
fn f<T>(x: T) -> T where T: Neg<Output=T> { -x }
```

in Rust.

Both allow us to call `f(2)` and `f(2.0)` etc.

Types

Overloading/Polymorphism

The programmer writes code just once, defining a function that will work on many types `T`. Superficially this looks like polymorphism: we can call `f(2)` and `f(2.0)` and the "same" code gets executed

Types

Overloading/Polymorphism

The programmer writes code just once, defining a function that will work on many types `T`. Superficially this looks like polymorphism: we can call `f(2)` and `f(2.0)` and the “same” code gets executed

But not really. The compiler simply writes for itself the code for the individual `int` and `double` versions and compiles those (or does the equivalent)

Types

Overloading/Polymorphism

The programmer writes code just once, defining a function that will work on many types `T`. Superficially this looks like polymorphism: we can call `f(2)` and `f(2.0)` and the "same" code gets executed

But not really. The compiler simply writes for itself the code for the individual `int` and `double` versions and compiles those (or does the equivalent)

```
int f(int x) { return -x; }
double f(double x) { return -x; }
```

Types

Overloading/Polymorphism

This approach is called *monomorphization*: replacing something apparently polymorphic with multiple monomorphic bits of code

Types

Overloading/Polymorphism

This approach is called *monomorphization*: replacing something apparently polymorphic with multiple monomorphic bits of code

And this is actually overloading `f` as the underlying code to negate an integer is different from the code to negate a floating point value

Types

Overloading/Polymorphism

This approach is called *monomorphization*: replacing something apparently polymorphic with multiple monomorphic bits of code

And this is actually overloading `f` as the underlying code to negate an integer is different from the code to negate a floating point value

And it would do the usual internal renaming

```
int f_int(int x) { return -x; }
double f_double(double x) { return -x; }
```

Types

Overloading/Polymorphism

This approach is called *monomorphization*: replacing something apparently polymorphic with multiple monomorphic bits of code

And this is actually overloading `f` as the underlying code to negate an integer is different from the code to negate a floating point value

And it would do the usual internal renaming

```
int f_int(int x) { return -x; }
double f_double(double x) { return -x; }
```

Exercise Make sure you understand why negation of integers is different code to negation of floating point

Types

Overloading/Polymorphism

Be aware that some people classify overloading as a particular kind of polymorphism, even though overloading uses different pieces of code for each type

Types

Overloading/Polymorphism

Be aware that some people classify overloading as a particular kind of polymorphism, even though overloading uses different pieces of code for each type

For them, the fact that two functions have the same name is enough to call it polymorphism

Types

Overloading/Polymorphism

Be aware that some people classify overloading as a particular kind of polymorphism, even though overloading uses different pieces of code for each type

For them, the fact that two functions have the same name is enough to call it polymorphism

Perhaps they are thinking of overloading the *name*, rather than overloading the *function*?

Types

Overloading/Polymorphism

Be aware that some people classify overloading as a particular kind of polymorphism, even though overloading uses different pieces of code for each type

For them, the fact that two functions have the same name is enough to call it polymorphism

Perhaps they are thinking of overloading the *name*, rather than overloading the *function*?

They call it *ad hoc polymorphism*, in contrast with true polymorphism, *parametric polymorphism*

overloading	$\leftrightarrow$	ad hoc polymorphism
polymorphism	$\leftrightarrow$	parametric polymorphism

Types

Overloading Return Types

Many languages only support overloading on function argument types, while conceivably you could overload on return types:

```
int f(int n) { ... }
double f(int n) { ... }
```

where we distinguish using the return type

Types

Overloading Return Types

Many languages only support overloading on function argument types, while conceivably you could overload on return types:

```
int f(int n) { ... }
double f(int n) { ... }
```

where we distinguish using the return type

This is much rarer

Types

Overloading Return Types

For example

```
int f(int n) { ... }
double f(int n) { ... }
int g(int n) { ... }
int g(double n) { ... }
```

where we overload `g` in the normal way

Types

Overloading Return Types

For example

```
int f(int n) { ... }
double f(int n) { ... }
int g(int n) { ... }
int g(double n) { ... }
```

where we overload `g` in the normal way

What should we do with `g(f(1))`?

Types

Overloading Return Types

For example

```
int f(int n) { ... }
double f(int n) { ... }
int g(int n) { ... }
int g(double n) { ... }
```

where we overload `g` in the normal way

What should we do with `g(f(1))`?

Overloading *both* argument types and return types is tricky: so we pick just one, and overloading arguments is generally more useful

Types

Overloading Return Types

Java and C++ don't support overloading on return types: so you can't have both `int foo(int)` and `double foo(int)`

Types

Overloading Return Types

Java and C++ don't support overloading on return types: so you can't have both `int foo(int)` and `double foo(int)`

You *can* have both `int foo(int)` and `double foo(double)` by virtue of the different argument types

Types

Overloading Return Types

Java and C++ don't support overloading on return types: so you can't have both `int foo(int)` and `double foo(int)`

You *can* have both `int foo(int)` and `double foo(double)` by virtue of the different argument types

Exercise Language with more sophisticated type systems, such as Rust and Haskell, do allow a form of overloading on return types. Read about this

Types

Overloading/Polymorphism

Monomorphization is not the only way a language might choose to implement polymorphism

Monomorphization is not the only way a language might choose to implement polymorphism

Exercise See *generics* in Java: this uses *Type Erasure* (which is actually parametric polymorphism)

Exercise See *generics* in Go: this uses a partial monomorphization technique called *GCShape stenciling with Dictionaries*

Types

Overloading/Polymorphism

Exercise Swift is superficially similar to other languages, e.g.,

```
func min<T: Comparable>(x: T, y: T) -> T {
  return y < x ? y : x
}
```

but again, it does something different. Read about *Generic Specialization* (which is kind of dynamic)

Exercise And read about C#'s approach to monomorphization: *Lazy Monomorphisation*

Types

Advanced Exercise Compare these monomorphization techniques

Exercise Find out what overloading your favourite languages support, e.g., overloading based on numbers of arguments to a function: `int f(int a)` and `int f(int a, int b)`

Types

Subtype Polymorphism

Next we have *subtype polymorphism* which is the kind of polymorphism that arises when we define a function on a type and apply it to an instance of a subtype

Types

Subtype Polymorphism

Next we have *subtype polymorphism* which is the kind of polymorphism that arises when we define a function on a type and apply it to an instance of a subtype

Almost always seen in the context of classes, rather than just general types

Types

Subtype Polymorphism

Next we have *subtype polymorphism* which is the kind of polymorphism that arises when we define a function on a type and apply it to an instance of a subtype

Almost always seen in the context of classes, rather than just general types

Some languages do support *subtypes*, as opposed to *subclasses*, e.g., positive integers as a subtype of all integers, but this is not common

Types

Subtype Polymorphism

For example if you have a class `Animal` with a subclass `Cat`

Types

Subtype Polymorphism

For example if you have a class `Animal` with a subclass `Cat`

A method defined on `Animal` will work on an instance of `Cat` even though they are not the same types

Types

Subtype Polymorphism

For example if you have a class `Animal` with a subclass `Cat`

A method defined on `Animal` will work on an instance of `Cat` even though they are not the same types

To emphasise this point: `Cat` and `Animal` are *different* classes, as you can't use them interchangeably

Types

Subtype Polymorphism

For example if you have a class `Animal` with a subclass `Cat`

A method defined on `Animal` will work on an instance of `Cat` even though they are not the same types

To emphasise this point: `Cat` and `Animal` are *different* classes, as you can't use them interchangeably

So this looks like a kind of polymorphism: a method working on multiple types

Types

Subtype Polymorphism

For example if you have a class `Animal` with a subclass `Cat`

A method defined on `Animal` will work on an instance of `Cat` even though they are not the same types

To emphasise this point: `Cat` and `Animal` are *different* classes, as you can't use them interchangeably

So this looks like a kind of polymorphism: a method working on multiple types

But subtype polymorphism — something every OO programmer relies on every day — is not actually different from the kinds of polymorphism we have seen already

Types

Subtype Polymorphism

Suppose we have classes

```
class Animal {
  bool alive() { ... }
  bool sleepy() { return false; }
}

class Cat extends Animal {
  bool sleepy() { return true; }
}
```

where `Cat` inherits the `alive` method but overrides the `sleepy` method

The `alive` method is parametric polymorphic: the same method works on more than one type, namely `Animal` and `Cat`

Types

Subtype Polymorphism

The `alive` method is parametric polymorphic: the same method works on more than one type, namely `Animal` and `Cat`

The `sleepy` method is ad-hoc polymorphic (overloaded) as we have two different bits of code with the same name, `sleepy`

The `alive` method is parametric polymorphic: the same method works on more than one type, namely `Animal` and `Cat`

The `sleepy` method is ad-hoc polymorphic (overloaded) as we have two different bits of code with the same name, `sleepy`

Thus “subtype polymorphic” is actually just a shorthand for “either ad-hoc or parametric polymorphic”

Types

Subtype Polymorphism

While talking about subtype polymorphism we should mention the *Liskov substitution principle*

Types

Subtype Polymorphism

While talking about subtype polymorphism we should mention the *Liskov substitution principle*

A principle that outlines the behaviour we should expect from subtyping

Types

Subtype Polymorphism

While talking about subtype polymorphism we should mention the *Liskov substitution principle*

A principle that outlines the behaviour we should expect from subtyping

> *Suppose* `S` *is a subtype of* `T`*. Then whenever we need an instance of type* `T` *we can use an instance of type* `S`*, and our code should still operate correctly*

While talking about subtype polymorphism we should mention the *Liskov substitution principle*

A principle that outlines the behaviour we should expect from subtyping

> *Suppose* `S` *is a subtype of* `T`*. Then whenever we need an instance of type* `T` *we can use an instance of type* `S`*, and our code should still operate correctly*

If this holds, instances of `S` really are instances of `T`, but perhaps with a few additional properties

Types

Subtype Polymorphism

While talking about subtype polymorphism we should mention the *Liskov substitution principle*

A principle that outlines the behaviour we should expect from subtyping

> *Suppose* `S` *is a subtype of* `T`*. Then whenever we need an instance of type* `T` *we can use an instance of type* `S`*, and our code should still operate correctly*

If this holds, instances of `S` really are instances of `T`, but perhaps with a few additional properties

Methods for `Animal` should work on `Cats`

This is most people's belief on how subtypes work: so why is it worth mentioning?

This is most people's belief on how subtypes work: so why is it worth mentioning?

Because some versions of inheritance and some uses of inheritance violate this principle

This is most people's belief on how subtypes work: so why is it worth mentioning?

Because some versions of inheritance and some uses of inheritance violate this principle

Some examples later, when we talk about *class composition*

Types

Note that the ideas of polymorphism and overloading are not reliant on OO: in fact they both predate OO

Types

Note that the ideas of polymorphism and overloading are not reliant on OO: in fact they both predate OO

As previously mentioned, a large number of languages overload the arithmetic functions like + and $*$, though most only in a fixed way

Types

Note that the ideas of polymorphism and overloading are not reliant on OO: in fact they both predate OO

As previously mentioned, a large number of languages overload the arithmetic functions like + and *, though most only in a fixed way

Lisp has always had parametric polymorphism (`length` of a list, etc.)

Types

Hacker Exercise C “supports” polymorphism using `void *`. Read about this

Exercise Ada supports subtyping, e.g., *integer ranges*, such as “integers 0…10” as a subtype of all integers. Read about this

Exercise We can also have polymorphic *datatypes*, e.g., `list` in Lisp, `struct Pair<T>(T, T)` in Rust, Java, and so on. Read about these, and determine whether they are parametric or ad-hoc