So (at last!) the way to choose a method for a given set of arguments is

1. find the CPLs for each argument
2. find all the applicable methods
3. sort the applicable methods in decreasing order of specificity according to the CPLs of the arguments
4. take the first (most specific) in the list

# Object Oriented Languages

## Method Dispatch

So (at last!) the way to choose a method for a given set of arguments is

1. find the CPLs for each argument
2. find all the applicable methods
3. sort the applicable methods in decreasing order of specificity according to the CPLs of the arguments
4. take the first (most specific) in the list

The sorted method list is useful for later when we want to be more inventive on using methods, such as method composition

Note this algorithm reduces to what we expect in the SI, single-dispatch case

Note this algorithm reduces to what we expect in the SI, single-dispatch case

**Exercise** Check this!

**Exercise** Given

```
(defgeneric foo (x y))
(defmethod foo ((x <number>) (y <number>)) 1)
(defmethod foo ((x <integer>) (y <integer>)) 2)
(defmethod foo ((x <number>) (y <float>)) 3)
(defmethod foo ((x <float>) (y <integer>)) 4)
```

work through the above algorithm to determine which method gets called on various arguments, such as `(foo 7 11)`, `(foo 7.0 11)`, `(foo 7.0 11.0)`, and so on.

This rather complex dispatch calculation will be done for *every* method call in your code

This rather complex dispatch calculation will be done for *every* method call in your code

Either at compile time (for a fixed class hierarchy), meaning no run-time overhead

# Object Oriented Languages

## Method Dispatch

This rather complex dispatch calculation will be done for *every* method call in your code

Either at compile time (for a fixed class hierarchy), meaning no run-time overhead

Or at run-time, meaning some considerable execution overhead

# Object Oriented Languages

## Method Dispatch

This rather complex dispatch calculation will be done for *every* method call in your code

Either at compile time (for a fixed class hierarchy), meaning no run-time overhead

Or at run-time, meaning some considerable execution overhead

. . . unless clever tricks are employed, e.g., caching

# Object Oriented Languages

## Method Dispatch

This rather complex dispatch calculation will be done for *every* method call in your code

Either at compile time (for a fixed class hierarchy), meaning no run-time overhead

Or at run-time, meaning some considerable execution overhead

. . . unless clever tricks are employed, e.g., caching

For example, a lot of effort has been put into JavaScript on precisely this point (and JavaScript is single dispatch!)

# Object Oriented Languages

## Multiple Inheritance

More on the question of multiple inheritance: languages have variations on MI. In C++

```
class D { ... }
class B: public D { ... }
class C: public D { ... }
class A: public B, public C { ... }
```

has the additional peculiarity that class `A` is defined to inherit *two* copies of `D`, one via `B` and one via `C`

# Object Oriented Languages

## Multiple Inheritance

More on the question of multiple inheritance: languages have variations on MI. In C++

```
class D { ... }
class B: public D { ... }
class C: public D { ... }
class A: public B, public C { ... }
```

has the additional peculiarity that class `A` is defined to inherit *two* copies of `D`, one via `B` and one via `C`

This is because occasionally we want two copies

# Object Oriented Languages

## Multiple Inheritance

More on the question of multiple inheritance: languages have variations on MI. In C++

```
class D { ... }
class B: public D { ... }
class C: public D { ... }
class A: public B, public C { ... }
```

has the additional peculiarity that class `A` is defined to inherit *two* copies of `D`, one via `B` and one via `C`

This is because occasionally we want two copies

For example, an `IOstream` inherits from both `Istream` and `Ostream`, which both inherit from `Stream`: we might want separate file pointers for input and output

# Object Oriented Languages

## Multiple Inheritance

If we want only a single copy, C++ requires what it calls *virtual inheritance*

```
class B: public virtual D { ... }
class C: public virtual D { ... }
class A: public B, public C { ... }
```

If we want only a single copy, C++ requires what it calls *virtual inheritance*

```
class B: public virtual D { ... }
class C: public virtual D { ... }
class A: public B, public C { ... }
```

Now the single copy of `D` is inherited by `A`

If we want only a single copy, C++ requires what it calls *virtual inheritance*

```
class B: public virtual D { ... }
class C: public virtual D { ... }
class A: public B, public C { ... }
```

Now the single copy of `D` is inherited by `A`

This is the most common usage, so should have been the default!

**Exercise** Find out how C++ addresses the diamond problem

**Exercise** Find out how Eiffel addresses the diamond problem

# Object Oriented Languages

## Multiple Inheritance

Java is one of several languages that avoid the complexities of MI not having it

Java is one of several languages that avoid the complexities of MI not having it

Instead, it supports *interfaces*

# Object Oriented Languages

## Multiple Inheritance

Java is one of several languages that avoid the complexities of MI not having it

Instead, it supports *interfaces*

As discussed in traits, this often just a list of method signatures, i.e., no code to go with the names

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

# Object Oriented Languages

## Multiple Inheritance

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

And the interface `Ostream` might name methods like `write` and `get_file_position`

# Object Oriented Languages

## Multiple Inheritance

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

And the interface `Ostream` might name methods like `write` and `get_file_position`

And then we might have

# Object Oriented Languages

## Multiple Inheritance

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

And the interface `Ostream` might name methods like `write` and `get_file_position`

And then we might have

```
class IOstream extends Stream
               implements Istream, Ostream {
...
}
```

# Object Oriented Languages

## Multiple Inheritance

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

And the interface `Ostream` might name methods like `write` and `get_file_position`

And then we might have

```
class IOstream extends Stream
               implements Istream, Ostream {
...
}
```

The class `IOstream` must implement — directly or inherited from (the single parent) `Stream` — all the methods mentioned in the definitions of interfaces `Istream` and `Ostream`

# Object Oriented Languages

## Multiple Inheritance

So, for this example, `IOstream` will possibly implement `read`, `write` and inherit `get_file_position`

# Object Oriented Languages

## Multiple Inheritance

So, for this example, `IOstream` will possibly implement `read`, `write` and inherit `get_file_position`

In Java, a class can derive from multiple interfaces, but not multiple classes

So, for this example, `IOstream` will possibly implement `read`, `write` and inherit `get_file_position`

In Java, a class can derive from multiple interfaces, but not multiple classes

There is no possibility of inheriting multiple methods of the same name, as the class can still only inherit a method from at most one parent class—and nothing from the interfaces

So, for this example, `IOstream` will possibly implement `read`, `write` and inherit `get_file_position`

In Java, a class can derive from multiple interfaces, but not multiple classes

There is no possibility of inheriting multiple methods of the same name, as the class can still only inherit a method from at most one parent class—and nothing from the interfaces

There is no problem with being told more than once that a class needs to implement a method of a given name and signature

So, for this example, `IOstream` will possibly implement `read`, `write` and inherit `get_file_position`

In Java, a class can derive from multiple interfaces, but not multiple classes

There is no possibility of inheriting multiple methods of the same name, as the class can still only inherit a method from at most one parent class—and nothing from the interfaces

There is no problem with being told more than once that a class needs to implement a method of a given name and signature

**Exercise** What happens if you derive from two interfaces and they ask for different signatures for the same method name?

An interface like this is more like a list of requirements for a class than about inheriting things

# Object Oriented Languages

## Multiple Inheritance

An interface like this is more like a list of requirements for a class than about inheriting things

But interfaces provide all of the MI functionality that most people need: they describe the required behaviour of a class, taking from multiple places

An interface like this is more like a list of requirements for a class than about inheriting things

But interfaces provide all of the MI functionality that most people need: they describe the required behaviour of a class, taking from multiple places

Without all the complexity of MI forcing some variety of inheritance on your code

# Object Oriented Languages

## Multiple Inheritance

An interface like this is more like a list of requirements for a class than about inheriting things

But interfaces provide all of the MI functionality that most people need: they describe the required behaviour of a class, taking from multiple places

Without all the complexity of MI forcing some variety of inheritance on your code

**Exercise** Later standards for Java allows default methods to be defined in interfaces, thus re-introducing the diamond problem. Find out how Java addresses this

# Object Oriented Languages

## Class Composition

Some people say MI is too complex, hard to implement properly, produces unexpected results, and can have performance issues, so you should not have it or use it

# Object Oriented Languages

## Class Composition

Some people say MI is too complex, hard to implement properly, produces unexpected results, and can have performance issues, so you should not have it or use it

They say if you want multiple behaviours, you can use SI with *class composition*

# Object Oriented Languages

## Class Composition

Some people say MI is too complex, hard to implement properly, produces unexpected results, and can have performance issues, so you should not have it or use it

They say if you want multiple behaviours, you can use SI with *class composition*

An `IOstream` should be a new, independent class, *containing* instances of `Istream` and `Ostream`

# Object Oriented Languages

## Class Composition

Some people say MI is too complex, hard to implement properly, produces unexpected results, and can have performance issues, so you should not have it or use it

They say if you want multiple behaviours, you can use SI with *class composition*

An `IOstream` should be a new, independent class, *containing* instances of `Istream` and `Ostream`

*Not* inheriting

# Object Oriented Languages

## Class Composition

Some people say MI is too complex, hard to implement properly, produces unexpected results, and can have performance issues, so you should not have it or use it

They say if you want multiple behaviours, you can use SI with *class composition*

An `IOstream` should be a new, independent class, *containing* instances of `Istream` and `Ostream`

*Not* inheriting

Note **class** composition is completely different from **method** composition!

# Object Oriented Languages

## Class Composition

```
class IOStream: public Istream, public Ostream { ... }
```

**inheriting** from `Istream` and `Ostream` becomes

```
class IOStream: { public: Istream i; Ostream o; ... }
```

**containing** `Istream` and `Ostream`

# Object Oriented Languages

## Class Composition

```
class IOStream: public Istream, public Ostream { ... }
```

**inheriting** from `Istream` and `Ostream` becomes

```
class IOStream: { public: Istream i; Ostream o; ... }
```

**containing** `Istream` and `Ostream`

And we need to write `str.i.ptr` or `str.o.ptr` as appropriate to get the stream pointers

# Object Oriented Languages

## Class Composition

```
class IOStream: public Istream, public Ostream { ... }
```

**inheriting** from `Istream` and `Ostream` becomes

```
class IOStream: { public: Istream i; Ostream o; ... }
```

**containing** `Istream` and `Ostream`

And we need to write `str.i.ptr` or `str.o.ptr` as appropriate to get the stream pointers

This can be used by SI languages, too, such as Java

# Object Oriented Languages

## Class Composition

We lose the convenience of the compiler doing automatic inheritance and automatic method selection, but many people argue multiple inheritance is too problematic to use correctly anyway

# Object Oriented Languages

## Class Composition

We lose the convenience of the compiler doing automatic inheritance and automatic method selection, but many people argue multiple inheritance is too problematic to use correctly anyway

So class composition is much more like delegation and prototyping OO

We lose the convenience of the compiler doing automatic inheritance and automatic method selection, but many people argue multiple inheritance is too problematic to use correctly anyway

So class composition is much more like delegation and prototyping OO

And we can combine behaviours arbitrarily, not being confined to a hierarchy

We lose the convenience of the compiler doing automatic inheritance and automatic method selection, but many people argue multiple inheritance is too problematic to use correctly anyway

So class composition is much more like delegation and prototyping OO

And we can combine behaviours arbitrarily, not being confined to a hierarchy

And no diamond problem

Many people say not to use MI as it has problems

Many people say not to use MI as it has problems

But that doesn't mean that *all* inheritance is problematic!

# Object Oriented Languages

## Class Composition

Many people say not to use MI as it has problems

But that doesn’t mean that *all* inheritance is problematic!

A lot of code successfully uses single inheritance

Many people say not to use MI as it has problems

But that doesn’t mean that *all* inheritance is problematic!

A lot of code successfully uses single inheritance

And a fair amount of code successfully uses MI!

But, of course, some people go further and say you should not even be using single inheritance, but should use class composition for everything

# Object Oriented Languages

## Class Composition

But, of course, some people go further and say you should not even be using single inheritance, but should use class composition for everything

A class inheritance hierarchy makes you share both structure (slots) and behaviour (methods), and you always get both when you inherit from a class

# Object Oriented Languages

## Class Composition

But, of course, some people go further and say you should not even be using single inheritance, but should use class composition for everything

A class inheritance hierarchy makes you share both structure (slots) and behaviour (methods), and you always get both when you inherit from a class

We don’t always want both: example shortly

Composition has several claimed downsides:

# Object Oriented Languages

## Multiple Inheritance/Class Composition

Composition has several claimed downsides:

- Lack of code reuse: but composition is also a way of avoiding code re-implementation, a kind of hand-crafted inheritance. Code is written once and reused

# Object Oriented Languages

## Multiple Inheritance/Class Composition

Composition has several claimed downsides:

- Lack of code reuse: but composition is also a way of avoiding code re-implementation, a kind of hand-crafted inheritance. Code is written once and reused
- Runtime overheads: same as inheritance, which can mean none if the compiler can statically determine which method to call

# Object Oriented Languages

## Multiple Inheritance/Class Composition

Composition has several claimed downsides:

- Lack of code reuse: but composition is also a way of avoiding code re-implementation, a kind of hand-crafted inheritance. Code is written once and reused
- Runtime overheads: same as inheritance, which can mean none if the compiler can statically determine which method to call
- Initialising an instance harder: composition calls (super)constructors, but so does inheritance

# Object Oriented Languages

## Multiple Inheritance/Class Composition

- Method/slot lookup: done at coding time, by the programmer, rather than by the compiler (recall `str.i.ptr` or `str.o.ptr` above), only a problem if you care about getting the right slot value

# Object Oriented Languages

## Multiple Inheritance/Class Composition

- Method/slot lookup: done at coding time, by the programmer, rather than by the compiler (recall `str.i.ptr` or `str.o.ptr` above), only a problem if you care about getting the right slot value
- Perhaps some lose a bit of encapsulation as “subclasses” need to be accessible

# Object Oriented Languages

## Multiple Inheritance/Class Composition

- Method/slot lookup: done at coding time, by the programmer, rather than by the compiler (recall `str.i.ptr` or `str.o.ptr` above), only a problem if you care about getting the right slot value
- Perhaps some lose a bit of encapsulation as “subclasses” need to be accessible
- Always get multiple versions of a slot, so composition is better with just behaviour (like traits/mixins!)

# Object Oriented Languages

## Multiple Inheritance/Class Composition

- Method/slot lookup: done at coding time, by the programmer, rather than by the compiler (recall `str.i.ptr` or `str.o.ptr` above), only a problem if you care about getting the right slot value
- Perhaps some lose a bit of encapsulation as “subclasses” need to be accessible
- Always get multiple versions of a slot, so composition is better with just behaviour (like traits/mixins!)

Of course, in contrast, inheritance can only adhere to the existing hierarchy, composition is not restricted

# Object Oriented Languages

## Liskov

We know MI has problems with inheritance

# Object Oriented Languages

## Liskov

We know MI has problems with inheritance

But so does SI: the world is not arranged in a nice neat hierarchy

We know MI has problems with inheritance

But so does SI: the world is not arranged in a nice neat hierarchy

Recall the Liskov substitution principle, a property we want from OO and inheritance:

> *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*

# Object Oriented Languages

## Liskov

We know MI has problems with inheritance

But so does SI: the world is not arranged in a nice neat hierarchy

Recall the Liskov substitution principle, a property we want from OO and inheritance:

> *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*

Here is an example where it goes wrong: or, rather, where inheritance is not helpful

The *circle-ellipse problem* (also known as the *square-rectangle problem*)

The *circle-ellipse problem* (also known as the *square-rectangle problem*)

Is a circle an ellipse with special properties?

The *circle-ellipse problem* (also known as the *square-rectangle problem*)

Is a circle an ellipse with special properties?

Or is an ellipse a circle with extra properties?

The *circle-ellipse problem* (also known as the *square-rectangle problem*)

Is a circle an ellipse with special properties?

Or is an ellipse a circle with extra properties?

The way you might use inheritance depends on your point of view

# Object Oriented Languages

## Circle-Ellipse

A circle is a special ellipse

```
Class ellipse {
      double rx, ry;
      ellipse(double x, double y) {... rx=x; ry=y...}
      void scale_x(double s) { rx = rx*s; }
      void scale_y(double s) { ry = ry*s; }
      double area() { return Pi*rx*ry; }
}

// a special ellipse where radii are equal
Class circle extends ellipse {
      circle(x: double) {... rx=x; ry=x; ...}
}
```

Circle inherits the `scale_x` method: what should it do?

- Should it be overridden to scale `ry` as well to maintain the constraint on the axes? Then scaling `x` by 2 unexpectedly quadruples the area, not doubles

Circle inherits the `scale_x` method: what should it do?

- Should it be overridden to scale `ry` as well to maintain the constraint on the axes? Then scaling `x` by 2 unexpectedly quadruples the area, not doubles
- Should the method be inapplicable — breaking Liskov?

Just scaling `x` alone breaks the requirement that the two radii in a circle are equal: a `circle` with different `rx` and `ry`

Just scaling `x` alone breaks the requirement that the two radii in a circle are equal: a `circle` with different `rx` and `ry`

And note that `ellipse(1.0,1.0)` is actually of a different class to `circle(1.0)`

# Object Oriented Languages

## Circle-Ellipse

Just scaling `x` alone breaks the requirement that the two radii in a circle are equal: a `circle` with different `rx` and `ry`

And note that `ellipse(1.0,1.0)` is actually of a different class to `circle(1.0)`

Exceptionally, some languages (e.g., Common Lisp) can change the class of an object: you can code things so that if you scale an instance of `circle` it becomes an instance of `ellipse`

# Object Oriented Languages

## Circle-Ellipse

Just scaling `x` alone breaks the requirement that the two radii in a circle are equal: a `circle` with different `rx` and `ry`

And note that `ellipse(1.0,1.0)` is actually of a different class to `circle(1.0)`

Exceptionally, some languages (e.g., Common Lisp) can change the class of an object: you can code things so that if you scale an instance of `circle` it becomes an instance of `ellipse`

But this is very rare: most languages don't do this

# Object Oriented Languages

## Circle-Ellipse

Alternatively, an ellipse a generalised circle

```
Class circle {
  double radius;
  circle(double x) {...r=x;...}
  double area() { return pi*radius*radius; }
}

Class ellipse extends circle {
  double radius2;
  ellipse(double x, double y) {... radius=x; radius2=y ...}
  scale_x(double s) { radius = radius*s; }
  scale_y(double s) { radius2 = radius2*s; }
  double area() { return pi*radius*radius2; }
}
```

# Object Oriented Languages

## Circle-Ellipse

This is not a natural use of inheritance, as ellipses don’t really have a radius

This is not a natural use of inheritance, as ellipses don't really have a radius

`radius` doesn't naturally correspond uniquely to one of `rx` or `ry`

This is not a natural use of inheritance, as ellipses don't really have a radius

`radius` doesn't naturally correspond uniquely to one of `rx` or `ry`

And we have to override all `circle` methods that depend on the radius

This is not a natural use of inheritance, as ellipses don't really have a radius

`radius` doesn't naturally correspond uniquely to one of `rx` or `ry`

And we have to override all `circle` methods that depend on the radius

Thus losing most of the benefit of inheritance

Liskov asks: can we can we use an instance of the subclass with every method for the superclass?

Liskov asks: can we can we use an instance of the subclass with every method for the superclass?

A circle is a special ellipse: possibly not, e.g., using `scale_x`

Liskov asks: can we can we use an instance of the subclass with every method for the superclass?

A circle is a special ellipse: possibly not, e.g., using `scale_x`

An ellipse a generalised circle: maybe, if we override most circle methods, so not really using inheritance

Even though we feel that circles and ellipses have some sort of relationship, we can’t capture that well using inheritance

Even though we feel that circles and ellipses have some sort of relationship, we can't capture that well using inheritance

Alternatives include class composition: e.g., `ellipse` contains `circle`

# Object Oriented Languages

## Circle-Ellipse

Even though we feel that circles and ellipses have some sort of relationship, we can't capture that well using inheritance

Alternatives include class composition: e.g., `ellipse` contains `circle`

Though this isn't much better than inheritance

# Object Oriented Languages

## Circle-Ellipse

We could use traits:

```
Class circle { double radius; }
Class ellipse { double rx, ry; }

trait Area { ... }
impl Area for circle { ... }
impl Area for ellipse { ... }

trait ScaleX { ... }
impl ScaleX for ellipse { ... }
// no ScaleX for circle
```

This works as traits separate behaviour from the classes, but there is no code sharing going on here (maybe there *can't* be any sharing?)

# Object Oriented Languages

## Circle-Ellipse

Other “fixes”:

# Object Oriented Languages

## Circle-Ellipse

Other “fixes”:

- Make all instances constant, thus not modifiable

# Object Oriented Languages

## Circle-Ellipse

Other “fixes”:

- Make all instances constant, thus not modifiable
- so `scale_x` could return a new instance of `ellipse` or `circle` when called on either circles or ellipses, but this would require the implementer of `ellipse` to know the class extension `circle` will also exist

Other “fixes”:

- Make all instances constant, thus not modifiable
- so `scale_x` could return a new instance of `ellipse` or `circle` when called on either circles or ellipses, but this would require the implementer of `ellipse` to know the class extension `circle` will also exist

```
ellipse scale_x(double s)
{
  if (s*rx == ry) { return circle(ry); }
  else { return ellipse(s*rx,ry); }
}
```

# Object Oriented Languages

## Circle-Ellipse

- Only use `ellipse`: the common case of circles becomes more noisy and error-prone to code for

# Object Oriented Languages

## Circle-Ellipse

- Only use `ellipse`: the common case of circles becomes more noisy and error-prone to code for
- `scale_x` is not applicable, or returns an error, or an exception when called on a `circle`: breaking Liskov

# Object Oriented Languages

## Circle-Ellipse

- Only use `ellipse`: the common case of circles becomes more noisy and error-prone to code for
- `scale_x` is not applicable, or returns an error, or an exception when called on a `circle`: breaking Liskov
- Change the hierarchy to have, say, `RoundObject` containing the commonality and `circle` and `ellipse` being sibling subclasses of `RoundObject`: but rewriting an existing hierarchy is not always possible; and the commonality might be less than you think

# Object Oriented Languages

## Circle-Ellipse

- Only use `ellipse`: the common case of circles becomes more noisy and error-prone to code for
- `scale_x` is not applicable, or returns an error, or an exception when called on a `circle`: breaking Liskov
- Change the hierarchy to have, say, `RoundObject` containing the commonality and `circle` and `ellipse` being sibling subclasses of `RoundObject`: but rewriting an existing hierarchy is not always possible; and the commonality might be less than you think
- Or just have no OO relationship between the two!

# Object Oriented Languages

## Choose the Right Tool

We are at the point that inheritance is not helping us

# Object Oriented Languages

## Choose the Right Tool

We are at the point that inheritance is not helping us

We might end up making our code worse trying to squeeze it into the OO paradigm

# Object Oriented Languages

## Choose the Right Tool

We are at the point that inheritance is not helping us

We might end up making our code worse trying to squeeze it into the OO paradigm

So OO is the wrong tool in such cases. Find a different approach

# Object Oriented Languages

## Choose the Right Tool

**Exercise** For games programmers: read about the *entity-component-system* (ECS) design pattern that favours composition over inheritance, used for at least the last two decades in games engines

**Exercise** React, the JavaScript platform for building UIs, is not OO, but is reasonably functional in style. Read about it