Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object

Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object

Compiled to machine code: C, C++, Fortran, . . .

Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object

Compiled to machine code: C, C++, Fortran, . . .

Bytecode: compile to a machine-independent code that is then interpreted or further compiled to machine code. Java, C#, Perl, Lua, Forth, . . .

Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object

Compiled to machine code: C, C++, Fortran, . . .

Bytecode: compile to a machine-independent code that is then interpreted or further compiled to machine code. Java, C#, Perl, Lua, Forth, . . .

Interpreted: Basic, HTML, . . .

Interpreted and Compiled

Feet

- C#: You forget precisely how to use the .NET interface and shoot yourself in the foot. You sue Microsoft for damages

Interpreted and Compiled

Feet

- C#: You forget precisely how to use the .NET interface and shoot yourself in the foot. You sue Microsoft for damages
- C# (2): You copy how Java shot itself in the foot. Then you explain to everybody who will listen how you did it better

Interpreted and Compiled

Feet

- C#: You forget precisely how to use the .NET interface and shoot yourself in the foot. You sue Microsoft for damages
- C# (2): You copy how Java shot itself in the foot. Then you explain to everybody who will listen how you did it better
- C# (3): You can create and shoot a gun in C#, but you can't shoot your foot in managed code

Interpreted and Compiled

Feet

- Lua: You come up with a decent way to shoot yourself in the foot, but you're unsure if it's the optimal way to go about it. You ask the mailing list. Someone points out that Lua has a "shoot foot" function built in, but it's only exposed via the C API. The discussion devolves into a long debate about whether various functions should be exposed, how objects and OOP should be implemented, and whether nil should be a valid table index

Interpreted and Compiled

Feet

- Lua: You come up with a decent way to shoot yourself in the foot, but you're unsure if it's the optimal way to go about it. You ask the mailing list. Someone points out that Lua has a “shoot foot” function built in, but it's only exposed via the C API. The discussion devolves into a long debate about whether various functions should be exposed, how objects and OOP should be implemented, and whether nil should be a valid table index
- Lua (2): You shoot yourself in the foot while watching enviously how Scheme shoots you in the foot

Interpreted and Compiled

Compiling to code for a specific processor produces fast running programs, using all the facilities of the hardware (when done properly)

Interpreted and Compiled

Compiling to code for a specific processor produces fast running programs, using all the facilities of the hardware (when done properly)

Provides lots of error checking in the compilation phase

Interpreted and Compiled

Compiling to code for a specific processor produces fast running programs, using all the facilities of the hardware (when done properly)

Provides lots of error checking in the compilation phase

Uses the compile-run-edit cycle of development

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

A kind of machine code for a standardised, *virtual machine*

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

A kind of machine code for a standardised, *virtual machine*

A real machine can then interpret the bytecode to run the program

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

A kind of machine code for a standardised, *virtual machine*

A real machine can then interpret the bytecode to run the program

Or compile the bytecode to native machine code and run that

Interpreted and Compiled

Bytecode produces more compact code and is machine independent so allowing mobile code

Interpreted and Compiled

Bytecode produces more compact code and is machine independent so allowing mobile code

Even mobile in the sense the program can move between different processors while it is running

Interpreted and Compiled

Bytecode produces more compact code and is machine independent so allowing mobile code

Even mobile in the sense the program can move between different processors while it is running

“Compile once and run anywhere”

Interpreted and Compiled

Bytecode produces more compact code and is machine independent so allowing mobile code

Even mobile in the sense the program can move between different processors while it is running

“Compile once and run anywhere”

Provides lots of error checking in the compilation phase

Interpreted and Compiled

Bytecode produces more compact code and is machine independent so allowing mobile code

Even mobile in the sense the program can move between different processors while it is running

“Compile once and run anywhere”

Provides lots of error checking in the compilation phase

Requires a separate run-time (the virtual machine) to interpret or further compile the code

Interpreted and Compiled

Bytecode produces more compact code and is machine independent so allowing mobile code

Even mobile in the sense the program can move between different processors while it is running

“Compile once and run anywhere”

Provides lots of error checking in the compilation phase

Requires a separate run-time (the virtual machine) to interpret or further compile the code

Generally a modest overhead in loss of speed in the execution of the bytecode

Managed and Unmanaged

Closely related is the idea of a *managed* language

Managed and Unmanaged

Closely related is the idea of a *managed* language

This produces code (often byte-compiled like Java and C#, but not exclusively) that only runs under a run-time virtual machine, not natively

Managed and Unmanaged

Closely related is the idea of a *managed* language

This produces code (often byte-compiled like Java and C#, but not exclusively) that only runs under a run-time virtual machine, not natively

The run-time then manages memory, usually including a GC, and does security checking, e.g., on network connections

Managed and Unmanaged

Closely related is the idea of a *managed* language

This produces code (often byte-compiled like Java and C#, but not exclusively) that only runs under a run-time virtual machine, not natively

The run-time then manages memory, usually including a GC, and does security checking, e.g., on network connections

The idea that this is a “safe” language, running in a secure *sandbox*, preventing all kinds of nasty things from happening: memory overruns, execution of virus code, connecting to rogue Web sites, and so on

Managed and Unmanaged

The idea extends to *managed data*, where (some or all of) the data is managed

Managed and Unmanaged

The idea extends to *managed data*, where (some or all of) the data is managed

For example (an extension to) C++ allows objects to be managed or unmanaged

Managed and Unmanaged

The idea extends to *managed data*, where (some or all of) the data is managed

For example (an extension to) C++ allows objects to be managed or unmanaged

Inaccurately and misleadingly, but to a decent approximation

$$\begin{aligned} \text{managed} &= \text{bytecode} \\ \text{unmanaged} &= \text{compiled} \end{aligned}$$

and the word “managed” is mostly used to make “unmanaged” sound bad by comparison

Interpreted and Compiled

Next are interpreted languages. This is good for rapid development where you don’t want to keep waiting for the compiler

Interpreted and Compiled

Next are interpreted languages. This is good for rapid development where you don't want to keep waiting for the compiler

No code as such, only source, so quite compact, but also requires a separate run-time interpreter

Interpreted and Compiled

Next are interpreted languages. This is good for rapid development where you don’t want to keep waiting for the compiler

No code as such, only source, so quite compact, but also requires a separate run-time interpreter

Large overhead in loss of speed as each line of code has to be interpreted before it can be executed

Interpreted and Compiled

Note: any given language can be compiled/interpreted/run in any of these ways

Interpreted and Compiled

Note: any given language can be compiled/interpreted/run in any of these ways

Though languages do tend to have a preferred approach

Interpreted and Compiled

Note: any given language can be compiled/interpreted/run in any of these ways

Though languages do tend to have a preferred approach

For example, C is almost always compiled, while Basic tends to be interpreted

Interpreted and Compiled

Java is bytecode, and has separate compile and runtime systems

Interpreted and Compiled

Java is bytecode, and has separate compile and runtime systems

Its main objective is machine-independent code

Interpreted and Compiled

Java is bytecode, and has separate compile and runtime systems

Its main objective is machine-independent code

Perl is bytecode and has an integrated compile and runtime system

Interpreted and Compiled

Java is bytecode, and has separate compile and runtime systems

Its main objective is machine-independent code

Perl is bytecode and has an integrated compile and runtime system

Each time a Perl program (the source text) is run it is first compiled, then executed: this helps rapid development, as above

Interpreted and Compiled

Java is bytecode, and has separate compile and runtime systems

Its main objective is machine-independent code

Perl is bytecode and has an integrated compile and runtime system

Each time a Perl program (the source text) is run it is first compiled, then executed: this helps rapid development, as above

Generally, the compiled form of the Perl program is not kept around

Interpreted and Compiled

Java is bytecode, and has separate compile and runtime systems

Its main objective is machine-independent code

Perl is bytecode and has an integrated compile and runtime system

Each time a Perl program (the source text) is run it is first compiled, then executed: this helps rapid development, as above

Generally, the compiled form of the Perl program is not kept around

Lua is similar to Perl in these respects

Interpreted and Compiled

Again, a non exclusive classification: some languages can be both interpreted and compiled, sometimes mixed within the same program (e.g., Lisp)

Interpreted and Compiled

Again, a non exclusive classification: some languages can be both interpreted and compiled, sometimes mixed within the same program (e.g., Lisp)

There exist Java to machine code compilers and C interpreters

Interpreted and Compiled

Again, a non exclusive classification: some languages can be both interpreted and compiled, sometimes mixed within the same program (e.g., Lisp)

There exist Java to machine code compilers and C interpreters

There are many hybrids: Cambridge CL compiles critical parts to C (thence machine code) for extra speed, but the rest is bytecode

Interpreted and Compiled

Again, a non exclusive classification: some languages can be both interpreted and compiled, sometimes mixed within the same program (e.g., Lisp)

There exist Java to machine code compilers and C interpreters

There are many hybrids: Cambridge CL compiles critical parts to C (thence machine code) for extra speed, but the rest is bytecode

This is important for targeting an application: compactness (for small machines) can be exchanged for raw speed of the running program. Or to allow mobility of the code

Interpreted and Compiled

Some systems initially interpret the program but keep note of those parts of code that are used frequently, e.g., loops

Interpreted and Compiled

Some systems initially interpret the program but keep note of those parts of code that are used frequently, e.g., loops

They then dynamically compile just those parts to machine code; the *Just in Time* (JIT) systems, e.g., in Java, JavaScript

Interpreted and Compiled

Some systems initially interpret the program but keep note of those parts of code that are used frequently, e.g., loops

They then dynamically compile just those parts to machine code; the *Just in Time* (JIT) systems, e.g., in Java, JavaScript

Others compile to interpreted bytecode, but then at runtime compile parts of the bytecode to machine code, as above

Interpreted and Compiled

Some systems initially interpret the program but keep note of those parts of code that are used frequently, e.g., loops

They then dynamically compile just those parts to machine code; the *Just in Time* (JIT) systems, e.g., in Java, JavaScript

Others compile to interpreted bytecode, but then at runtime compile parts of the bytecode to machine code, as above

Occasionally JIT can produce faster running code than simple static compilation as the compilation process can be informed by the profile information gained from running the program (e.g., which methods are actually being called)

Interpreted and Compiled

Though this does incur some overhead: compilation is not cheap, and unless you are careful it can dominate the running time in a short-lived program

Interpreted and Compiled

Though this does incur some overhead: compilation is not cheap, and unless you are careful it can dominate the running time in a short-lived program

You might argue that it doesn’t matter in a short-lived program as it’s done soon anyway. However if you run that program many times it does add up to a lot of extra CPU cycles (i.e., energy)

Interpreted and Compiled

Though this does incur some overhead: compilation is not cheap, and unless you are careful it can dominate the running time in a short-lived program

You might argue that it doesn’t matter in a short-lived program as it’s done soon anyway. However if you run that program many times it does add up to a lot of extra CPU cycles (i.e., energy)

Long-running programs benefit a lot, though

Interpreted and Compiled

Though this does incur some overhead: compilation is not cheap, and unless you are careful it can dominate the running time in a short-lived program

You might argue that it doesn't matter in a short-lived program as it's done soon anyway. However if you run that program many times it does add up to a lot of extra CPU cycles (i.e., energy)

Long-running programs benefit a lot, though

Exercise. Look at the optimisations that modern implementations of JavaScript use

Interpreted and Compiled

Another approach is *ahead of time* (AOT) compilation

Interpreted and Compiled

Another approach is *ahead of time* (AOT) compilation

This takes bytecode and further compiles it for the specific OS and hardware at *installation time*

Interpreted and Compiled

Another approach is *ahead of time* (AOT) compilation

This takes bytecode and further compiles it for the specific OS and hardware at *installation time*

Devised mostly for users (not developers!) of apps for low-energy devices (phones), where the repeated runtime interpretation or JIT compilation every time the app is run is wasted energy

Interpreted and Compiled

Another approach is *ahead of time* (AOT) compilation

This takes bytecode and further compiles it for the specific OS and hardware at *installation time*

Devised mostly for users (not developers!) of apps for low-energy devices (phones), where the repeated runtime interpretation or JIT compilation every time the app is run is wasted energy

Suitable compilation and optimisation is done just once, when the app is installed

Interpreted and Compiled

AOT gives us

Interpreted and Compiled

AOT gives us

- a faster running app, as there is no run-time overhead of interpretation or compilation

Interpreted and Compiled

AOT gives us

- a faster running app, as there is no run-time overhead of interpretation or compilation
- less energy used, as we don't repeatedly use energy in doing the same compilation every time the app is run

Interpreted and Compiled

Downsides include

Interpreted and Compiled

Downsides include

- you lose the run-time information of a JIT that could possibly produce better optimised code. However, this loss appears to be outweighed by the gains from being able to optimise globally the whole app, rather than JIT's local optimisations

Interpreted and Compiled

Downsides include

- you lose the run-time information of a JIT that could possibly produce better optimised code. However, this loss appears to be outweighed by the gains from being able to optimise globally the whole app, rather than JIT's local optimisations
- installing the app will take a lot longer if a thorough optimising compiler is used. A user would do this just once, though

Interpreted and Compiled

Downsides include

- you lose the run-time information of a JIT that could possibly produce better optimised code. However, this loss appears to be outweighed by the gains from being able to optimise globally the whole app, rather than JIT's local optimisations
- installing the app will take a lot longer if a thorough optimising compiler is used. A user would do this just once, though
- the compiled code takes up more space. Becoming less of an issue as memory capacity on small devices improves

Interpreted and Compiled

You can also use a mixture of AOT and JIT

Interpreted and Compiled

You can also use a mixture of AOT and JIT

Android Nougat does not use AOT when installing an app

Interpreted and Compiled

You can also use a mixture of AOT and JIT

Android Nougat does not use AOT when installing an app

When your phone is idle it then sneakily uses AOT while you are not looking

Interpreted and Compiled

You can also use a mixture of AOT and JIT

Android Nougat does not use AOT when installing an app

When your phone is idle it then sneakily uses AOT while you are not looking

And it also uses JIT to tune apps as they run

Interpreted and Compiled

You get the advantages of fast installation and AOT and JIT

Interpreted and Compiled

You get the advantages of fast installation and AOT and JIT

But this makes the Android runtime very complicated!

Interpreted and Compiled

Exercise. Look at several languages and determine their usual methods of execution

Exercise. Then determine the positives and negatives of doing it differently (e.g., compiling Java to machine code; bytecoding C)

Exercise. Another approach is for the app store to take the code and compile and pre-optimise it into separate codes for each of the various kinds of hardware out there. Then it delivers the appropriately optimised code at download time. Find out about this

Exercise. How is using AOT different from using a classical compiler?

Compilation

You may wish to think about how compilation affects optimisation of your code

Compilation

You may wish to think about how compilation affects optimisation of your code

“Normal” Compilation

A compiler is given a module/file at a time and compiles it, usually with some type information about the external functions called (e.g., `#include`, or `use` or equivalent)

Compilation

You may wish to think about how compilation affects optimisation of your code

“Normal” Compilation

A compiler is given a module/file at a time and compiles it, usually with some type information about the external functions called (e.g., `#include`, or `use` or equivalent)

So if the code includes a call `f(x+1,y/2)`, where `f` is defined in another module, the compiler generally only has the type signature `int f(int a, int b)` so it knows enough to generate the correct code to pass the arguments and return the value

Compilation

The code for `f` could be in a separate module, compiled at another time or place, so the compiler has no more information than the signature, and can make no assumptions on `f`

Compilation

The code for `f` could be in a separate module, compiled at another time or place, so the compiler has no more information than the signature, and can make no assumptions on `f`

E.g., if it knew that `b` was unused in `f`, it could optimise away the `y` and the division

Compilation

The code for `f` could be in a separate module, compiled at another time or place, so the compiler has no more information than the signature, and can make no assumptions on `f`

E.g., if it knew that `b` was unused in `f`, it could optimise away the `y` and the division

But without knowing more about `f`, it can't do anything clever like that

Compilation

Total Compilation

This is quite rare in practice, usually only for small programs

Compilation

Total Compilation

This is quite rare in practice, usually only for small programs

The compiler is given the *whole* program code at once

Compilation

Total Compilation

This is quite rare in practice, usually only for small programs

The compiler is given the *whole* program code at once

It can now look at every detail of every function and make optimisations such as the one above

Compilation

Total Compilation

This is quite rare in practice, usually only for small programs

The compiler is given the *whole* program code at once

It can now look at every detail of every function and make optimisations such as the one above

Practically, this is clearly quite difficult for larger programs

Compilation

Link Time Optimisation (LTO)

Modules are compiled separately as normal, but in the link phase, when all the compiled parts are joined together, the linker can make some optimisations

Compilation

Link Time Optimisation (LTO)

Modules are compiled separately as normal, but in the link phase, when all the compiled parts are joined together, the linker can make some optimisations

Again, technically difficult, but starting to make a big difference

Compilation

Run Time Optimisation

The runtime system monitors the program as it is running, and make dynamic optimisations to the code using knowledge of what is actually happening in the code

Compilation

Run Time Optimisation

The runtime system monitors the program as it is running, and make dynamic optimisations to the code using knowledge of what is actually happening in the code

This might involve moving bits of code or data around based on how often they are needed, to reduce memory pressure

Compilation

Run Time Optimisation

The runtime system monitors the program as it is running, and make dynamic optimisations to the code using knowledge of what is actually happening in the code

This might involve moving bits of code or data around based on how often they are needed, to reduce memory pressure

Used to good effect in JIT compilers

Classifications

There are a large number of ways we can look at languages

Classifications

There are a large number of ways we can look at languages

It is important to know that these classifications exist so we can make informed choices amongst them

Classifications

There are a large number of ways we can look at languages

It is important to know that these classifications exist so we can make informed choices amongst them

The right tool for the job

Classifications

There are a large number of ways we can look at languages

It is important to know that these classifications exist so we can make informed choices amongst them

The right tool for the job

We can also move knowledge between specific languages

Classifications

There are a large number of ways we can look at languages

It is important to know that these classifications exist so we can make informed choices amongst them

The right tool for the job

We can also move knowledge between specific languages

Learning a new language is not a problem: usually a matter of looking at a book to get the syntax

Classifications

There are a large number of ways we can look at languages

It is important to know that these classifications exist so we can make informed choices amongst them

The right tool for the job

We can also move knowledge between specific languages

Learning a new language is not a problem: usually a matter of looking at a book to get the syntax

There are only a few features unique to a language: these are the bits you should concentrate on

Classifications

There are a large number of ways we can look at languages

It is important to know that these classifications exist so we can make informed choices amongst them

The right tool for the job

We can also move knowledge between specific languages

Learning a new language is not a problem: usually a matter of looking at a book to get the syntax

There are only a few features unique to a language: these are the bits you should concentrate on

The rest is easy

Object Oriented Languages

We are now going to spend some more time looking at OO languages as they are important and have a wide variety of variants amongst themselves

Object Oriented Languages

We are now going to spend some more time looking at OO languages as they are important and have a wide variety of variants amongst themselves

It’s a big family

Object Oriented Languages

We are now going to spend some more time looking at OO languages as they are important and have a wide variety of variants amongst themselves

It's a big family

Many people have the implicit assumption that if you know Java then you know all about OO

Object Oriented Languages

We are now going to spend some more time looking at OO languages as they are important and have a wide variety of variants amongst themselves

It's a big family

Many people have the implicit assumption that if you know Java then you know all about OO

This is far from the truth: Java way of doing OO is just one way of many

Object Oriented Languages

It is sometimes said that an OO language has

> *“Abstraction, Encapsulation, Inheritance, Polymorphism”*

Object Oriented Languages

It is sometimes said that an OO language has

> *“Abstraction, Encapsulation, Inheritance, Polymorphism”*

We shall see the several ways that this is wrong!

Object Oriented Languages

Many people think that OO is about *classes*

Object Oriented Languages

Many people think that OO is about *classes*

And so say the first step in OO design is to map out your class hierarchy

Object Oriented Languages

Many people think that OO is about *classes*

And so say the first step in OO design is to map out your class hierarchy

This is also misleading: OO is actually about *objects*

Object Oriented Languages

Many people think that OO is about *classes*

And so say the first step in OO design is to map out your class hierarchy

This is also misleading: OO is actually about *objects*

Classes are secondary, and sometimes not there at all!

Object Oriented Languages

> *It was obvious to me 20-some years ago that OOP wasn't a panacea. That's the reason C++ supports several design and programming styles.*
>
> *In the first edition of "The C++ Programming Language," I didn't use the phrase "object-oriented programming" because I didn't want to feed the hype. One of the problems with OOP is exactly that unscrupulous people have hyped it as a panacea. Overselling something inevitably leads to disappointments.*

Bjarne Stroustrup, Feb 2000

Object Oriented Languages

Language historians put the emergence of the idea of objects and classes in a purpose-designed language perhaps as far back as 1962 with Simula, a discrete event simulation language, and more definitely in 1967 with Simula 67

Object Oriented Languages

Language historians put the emergence of the idea of objects and classes in a purpose-designed language perhaps as far back as 1962 with Simula, a discrete event simulation language, and more definitely in 1967 with Simula 67

Simula looks like a mixture of Pascal and Java, and has been described as “Algol plus classes”

Object Oriented Languages

Simula has constructs like objects, classes, subclasses and virtual methods that followed through C++ directly into Java

Object Oriented Languages

Simula has constructs like objects, classes, subclasses and virtual methods that followed through C++ directly into Java

> *C++ is Simula in wolf's clothing*

Bjarne Stroustrup

Object Oriented Languages

Simula has constructs like objects, classes, subclasses and virtual methods that followed through C++ directly into Java

> *C++ is Simula in wolf's clothing*

Bjarne Stroustrup

However, it was with Smalltalk in 1972 that the OO concept really took off

Object Oriented Languages

Feet

- Simula: ?

Object Oriented Languages

Feet

- Simula: ?
- Smalltalk: You send the message shoot to gun, with selectors bullet and myFoot. A window pops up saying Gunpowder doesNotUnderstand: spark. After several fruitless hours spent browsing the methods for Trigger, FiringPin and IdealGas, you take the easy way out and create ShotFoot, a subclass of Foot with an additional instance variable bulletHole

Object Oriented Languages

Reflection

Smalltalk introduced *metaclasses*, classes that determine the behaviour of other classes, thus enabling *reflection* in programs

Object Oriented Languages

Reflection

Smalltalk introduced *metaclasses*, classes that determine the behaviour of other classes, thus enabling *reflection* in programs

The concept of reflection, where a language can inspect and alter itself is dangerously close to the idea of self-modifying programs

Object Oriented Languages

Reflection

Smalltalk introduced *metaclasses*, classes that determine the behaviour of other classes, thus enabling *reflection* in programs

The concept of reflection, where a language can inspect and alter itself is dangerously close to the idea of self-modifying programs

Self-modifying programs are dangerous and hard to understand or control

Object Oriented Languages

Reflection

Smalltalk introduced *metaclasses*, classes that determine the behaviour of other classes, thus enabling *reflection* in programs

The concept of reflection, where a language can inspect and alter itself is dangerously close to the idea of self-modifying programs

Self-modifying programs are dangerous and hard to understand or control

But metaobject programming as a way to implement reflection puts a framework on this which makes it safe to use

Object Oriented Languages

Reflection

Smalltalk introduced *metaclasses*, classes that determine the behaviour of other classes, thus enabling *reflection* in programs

The concept of reflection, where a language can inspect and alter itself is dangerously close to the idea of self-modifying programs

Self-modifying programs are dangerous and hard to understand or control

But metaobject programming as a way to implement reflection puts a framework on this which makes it safe to use

But still very powerful

A related idea is *reification*

A related idea is *reification*

This is where a system can look at its own structure or behaviour

Object Oriented Languages

Reflection

A related idea is *reification*

This is where a system can look at its own structure or behaviour

Sometimes called *introspection*, and is often seen as making certain aspects first-class objects, for example first-class classes, or lambdas as a reification of functions

Object Oriented Languages

Reflection

A related idea is *reification*

This is where a system can look at its own structure or behaviour

Sometimes called *introspection*, and is often seen as making certain aspects first-class objects, for example first-class classes, or lambdas as a reification of functions

Debuggers can be viewed as reification; as can class-loaders in Java; and `eval` in Lisp

A related idea is *reification*

This is where a system can look at its own structure or behaviour

Sometimes called *introspection*, and is often seen as making certain aspects first-class objects, for example first-class classes, or lambdas as a reification of functions

Debuggers can be viewed as reification; as can class-loaders in Java; and `eval` in Lisp

Reflection is where the system can go in and modify things, too

Object Oriented Languages

In Smalltalk, everything is an object, including control structures like `if`

Object Oriented Languages

In Smalltalk, everything is an object, including control structures like `if`

And everything is mediated by messages sent between objects

Object Oriented Languages

In Smalltalk, everything is an object, including control structures like `if`

And everything is mediated by messages sent between objects

Even addition is a message: `2 + 3` is the syntax that sends the message + (with argument 3) to the object 2

Object Oriented Languages

In Smalltalk, everything is an object, including control structures like `if`

And everything is mediated by messages sent between objects

Even addition is a message: `2 + 3` is the syntax that sends the message + (with argument 3) to the object 2

There is no artificial separation of primitive objects from other objects like in Java

Object Oriented Languages

In Smalltalk, everything is an object, including control structures like `if`

And everything is mediated by messages sent between objects

Even addition is a message: `2 + 3` is the syntax that sends the message + (with argument 3) to the object 2

There is no artificial separation of primitive objects from other objects like in Java

This is more like `2.plus(3)` in Java-like syntax

Object Oriented Languages

Smalltalk prompted a lot of research into OO in the 70s and 80s

Object Oriented Languages

Smalltalk prompted a lot of research into OO in the 70s and 80s

And many different styles of OO were proposed including features called *prototyping* and *delegation*, and then Lisp-based languages featuring multiple inheritance and metaobject protocols

Object Oriented Languages

Smalltalk prompted a lot of research into OO in the 70s and 80s

And many different styles of OO were proposed including features called *prototyping* and *delegation*, and then Lisp-based languages featuring multiple inheritance and metaobject protocols

But we shall start with the most familiar kind of OO: that typified by having classes arranged in a hierarchy

Object Oriented Languages

Class Hierarchy

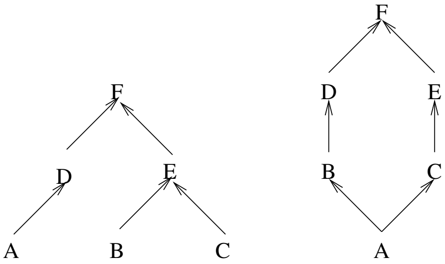
The *class hierarchy* is the relationship between classes

Object Oriented Languages

Class Hierarchy

The *class hierarchy* is the relationship between classes

This can be in a *graph*, where a class inherits from a single parent class; or a *directed acyclic graph* (DAG) when classes can inherit from more than one parent



A Graph and a DAG

Object Oriented Languages

Class Hierarchy

Both trees and DAGS have an important property: no loops

Object Oriented Languages

Class Hierarchy

Both trees and DAGS have an important property: no loops

A loop would entail a class inheriting (possibly indirectly) from itself

Object Oriented Languages

Class Hierarchy

Both trees and DAGS have an important property: no loops

A loop would entail a class inheriting (possibly indirectly) from itself

Thus we do not allow loops in the class hierarchy

Object Oriented Languages

Class Hierarchy

In some languages, e.g., Lisp and Smalltalk, classes are part of the runtime, being objects that can be manipulated in the program

Object Oriented Languages

Class Hierarchy

In some languages, e.g., Lisp and Smalltalk, classes are part of the runtime, being objects that can be manipulated in the program

In others, e.g., C++ and Java, classes are part of the program design but not first-class objects in the system

Object Oriented Languages

Class Hierarchy

In some languages, e.g., Lisp and Smalltalk, classes are part of the runtime, being objects that can be manipulated in the program

In others, e.g., C++ and Java, classes are part of the program design but not first-class objects in the system

Exercise. But look up `java.lang.reflect`

Object Oriented Languages

Class Hierarchy

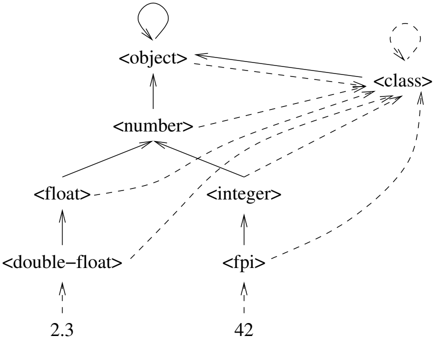
In some languages, e.g., Lisp and Smalltalk, classes are part of the runtime, being objects that can be manipulated in the program

In others, e.g., C++ and Java, classes are part of the program design but not first-class objects in the system

Exercise. But look up `java.lang.reflect`

A language will have a default hierarchy of those classes that come with the language

Object Oriented Languages



Part of the EuLisp Class Hierarchy (simplified)

There are *two* hierarchies in this diagram

Dotted arrow is *instance of*/*member of*/*is a*; solid arrow is *inherits from*/*subclass*/*extends*/*subset*

Object Oriented Languages

Every object is an *instance* of a class (dotted arrow), and is sometimes called a *member* of that class.

Object Oriented Languages

Every object is an *instance* of a class (dotted arrow), and is sometimes called a *member* of that class.

E.g., the integer 42 is an instance of the class `<fpi>`

Object Oriented Languages

Every object is an *instance* of a class (dotted arrow), and is sometimes called a *member* of that class.

E.g., the integer 42 is an instance of the class `<fpi>`

E.g., the class `<fpi>` is an instance of the class `<class>`

Object Oriented Languages

Every object is an *instance* of a class (dotted arrow), and is sometimes called a *member* of that class.

E.g., the integer 42 is an instance of the class `<fpi>`

E.g., the class `<fpi>` is an instance of the class `<class>`

A *subclass* will *inherit* (solid arrow) from its parent *superclass* (or superclass*es*)

Object Oriented Languages

Every object is an *instance* of a class (dotted arrow), and is sometimes called a *member* of that class.

E.g., the integer 42 is an instance of the class `<fpi>`

E.g., the class `<fpi>` is an instance of the class `<class>`

A *subclass* will *inherit* (solid arrow) from its parent *superclass* (or superclass*es*)

It inherits both *structure*/*attributes* (how the instances are stored in memory); and *behaviour* (the methods)

Object Oriented Languages

Every object is an *instance* of a class (dotted arrow), and is sometimes called a *member* of that class.

E.g., the integer 42 is an instance of the class `<fpi>`

E.g., the class `<fpi>` is an instance of the class `<class>`

A *subclass* will *inherit* (solid arrow) from its parent *superclass* (or superclass*es*)

It inherits both *structure*/*attributes* (how the instances are stored in memory); and *behaviour* (the methods)

Of course, it may override or add to either: generally you override methods, but add to attributes

Object Oriented Languages

E.g., `<fpi>` inherits from `<integer>`

Object Oriented Languages

E.g., `<fpi>` inherits from `<integer>`

And `<class>` inherits from `<object>`

Object Oriented Languages

E.g., `<fpi>` inherits from `<integer>`

And `<class>` inherits from `<object>`

`<object>` inherits from itself

Object Oriented Languages

E.g., `<fpi>` inherits from `<integer>`

And `<class>` inherits from `<object>`

`<object>` inherits from itself

This is safe to do, as `<object>` has no structure or behaviour

Object Oriented Languages

E.g., `<fpi>` inherits from `<integer>`

And `<class>` inherits from `<object>`

`<object>` inherits from itself

This is safe to do, as `<object>` has no structure or behaviour

The class `<object>` is an instance of the class `<class>`

Object Oriented Languages

E.g., `<fpi>` inherits from `<integer>`

And `<class>` inherits from `<object>`

`<object>` inherits from itself

This is safe to do, as `<object>` has no structure or behaviour

The class `<object>` is an instance of the class `<class>`

Of course, the class `<class>` is an instance of itself

Object Oriented Languages

So there are two kinds of relationships between objects: instance and inherits

Object Oriented Languages

So there are two kinds of relationships between objects: instance and inherits

And two kinds of object: classes and non-classes

Object Oriented Languages

So there are two kinds of relationships between objects: instance and inherits

And two kinds of object: classes and non-classes

We can make instances of classes, but not of non-classes

Object Oriented Languages

So there are two kinds of relationships between objects: instance and inherits

And two kinds of object: classes and non-classes

We can make instances of classes, but not of non-classes

Other kinds of OO dispense with one or both of these relationships

Object Oriented Languages

So there are two kinds of relationships between objects: instance and inherits

And two kinds of object: classes and non-classes

We can make instances of classes, but not of non-classes

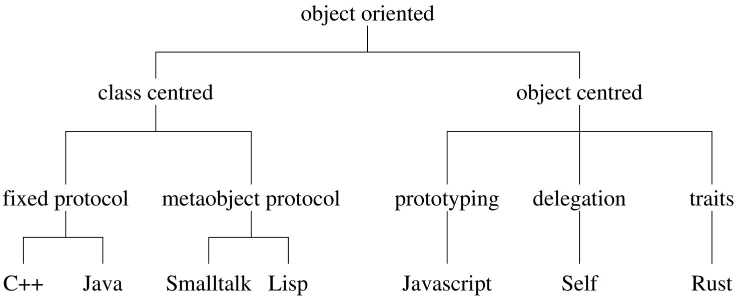
Other kinds of OO dispense with one or both of these relationships

Or one of these kinds of object: the classes

Object Oriented Languages

Exercise. For Java, C++, Common Lisp, EuLisp and any others determine their initial class hierarchy

Object Oriented Languages



object oriented

class centred

object centred

fixed protocol

metaobject protocol

prototyping

delegation

traits

C++

Java

Smalltalk

Lisp

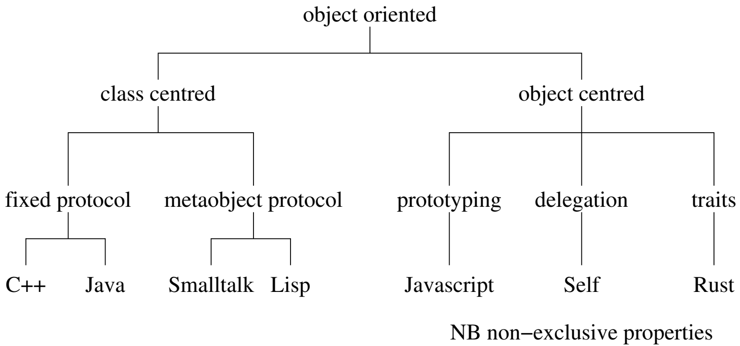
Javascript

Self

Rust

NB non–exclusive properties

Object Oriented Languages



Exercise. In this picture, determine which are instance links and which are inheritance links!