# MATH0027:
# Object Oriented Mechanisms

Russell Bradford
and
Julian Padget

## 1 Introduction

### 1.1 What is the course about?

This course follows on from MATH0019 (Computation, Formal Logic etc.). The connection may not become clear until later in the course, but these two courses explore

- the mathematical origins of programming languages
- the relative strengths and weaknesses of languages based on these principles.

This course examines another in the family of computer languages:

1. Procedural: of pragmatic origins, these are what most people think programming (and, indeed, computing) is all about. They have developed from the von Neumann model, and physical computer architectures. Examples include C and Fortran.
2. Functional: these come from $\lambda$-calculus and rewriting. Original examples are Lisp and Scheme; later types were added, as in ML and Haskell.
3. Logic: derived from logic and rewriting. The principle exponent of this family is Prolog.
4. Object-Oriented: of pragmatic origins, but with strong theoretical influences.

Of course, these families are not strictly independent: lots of interbreeding goes on.

Thus we are going to be looking at the last in the above list, building on what you know from the first and second.

### 1.2 Content

We will be looking at several languages. The major ones are

- C and C++
- Java
- Javascript
- Lisp and Scheme

Of course, C and Scheme are not object oriented (in the widely accepted sense), but we will be using these to demonstrate how OOLs are implemented.

By now, you should be able to read a program in most any language (excepting a few of the more quirky ones!) and get a good idea of its meaning. You should be prepared to write small programs in any of the above.

Notice that this course is *not* specifically about teaching you C++, or Java, or whatever. It *is* about the theory and mechanisms behind these languages. If you want to learn the deep and dirty details of writing C++programs, read a book. If you want to learn the principles of object oriented methods, so that you can choose between using C++, Java, etc., then this course is for you!

> Aside: Actually, the best way to learn a new language is to write lots of programs!

There is 25% coursework! This should appear approximately half-way through the course, and will include some programming.

The following syllabus is subject to arbitrary changes!

- So what is "object oriented"? Taxonomy: there are many types of OOL, and we need to sort out the relationships between them.
- Class centred vs. Object centred. The major divisions of types of OOL. Illustrated by implementations of complex numbers and sequences. This will take a fair number of lectures to cover.
- How do OO systems work? Behind the scenes of how the clever stuff that OOLs give you actually works. We will build OO systems from non-OO languages. This will explain more of the taxonomic tree.
- Multiple Inheritance and Reflection. More advanced stuff. Including questions like "is OO really the solution?", and how to pull yourself up by your bootstraps.

### 1.3 Resources

The literature on OO is huge. For a while the subject was very trendy, and thus generated a mass of books and papers. More heat than light, mostly.

> Aside: These days, as the problems and limitations of OO systems have become understood, the trendy researcher has moved on to newer pastures.

Specifically, for this course:

- "An Introduction to Object-Oriented Programming," 2nd-ed, Timothy Budd, Addison Wesley. A good generalist text.
- "Structure and Interpretation of Computer Programs," Abelson and Sussman. A good introduction to structured program design.
- "The Art of the Metaobject Protocol," Gregor Kiczales et al. An advanced text covering everything in OO design.
- "Smalltalk-80, The Language and its Implementation," Goldberg and Robinson. A good introduction to OO terms and concepts in one of the pioneer OO languages.

- "Classical Data Structures in C++," Timothy Budd. A better C++ book than most.
- "Data Structures in C++ using the Standard Template Library," Timothy Budd. A revision of the above using the newly-standardised STL.
- Julian Padget's Web page for MATH0027, `http://www.maths.bath.ac.uk/~masjap/MATH0027`

# 2 What is "Object Oriented"?

## 2.1 History

A little history first, for perspective. Each decade had a "big idea:"

- 1960s: high level languages (HLLs)
- 1970s: structured programming
- 1980s: $4^{\text{th}}$ generation languages
- 1990s: object oriented programming
- 2000s: ?

> Aside: The $4^{\text{th}}$ generation languages are application specific, e.g., PostScript, Maple, SQL, etc. The $5^{\text{th}}$ generation was the AI and logic languages that never quite took off.

But the history of OOLs starts much earlier than the 1990s. Since programming was invented, there has been a steady stream of "crises in the software industry," i.e., people writing papers on how programs have become too complex to write, maintain and understand. Many "solutions" have been offered, and OO is just one of them. OO did not appear ready-formed overnight, it has had a gestation of over 20 years.

- 1967: SIMULA, a development of Algol 60 that gave us classes
- 1972: Smalltalk, a "proper" OO language that gave us metaclasses
- 1970-75: Actor languages, a sequence of languages that gave us prototyping and delegation
- 1980-85: Flavors/Loops/Common Lisp Object System (CLOS), which gave us multiple inheritance and metaobject protocols.

These words will become imbued with meaning in this course!

All the above developments had the objective of

(a) controlling complexity, through the better management of the software development process, and

(b) reducing costs, through code reuse, and easier code maintenance through low coupling of modules.

Whether they were successful is very much open to debate.

## 2.2 What makes OO different?

You are familiar with the *procedural model* of programming, as is taught in classical software engineering. This tells us to break our code into lots of little functions which call each other. The idea is to capture an idea in a procedure, and that makes its easy to write the code and test it.

```
step1(arg, arg, ...);
step2(arg, arg, ...);
```

Data is passive, and all the action happens in the procedures. The tendency is for the procedures to become interdependent in complex programs, and changing one may necessitate changing many others.b Similarly, changing the format of the data will incur many changes in the procedures.

An object oriented systems treats data as as first class citizen and just as important as the procedures: in fact, more so. OO systems are based around the world view that of *objects that communicate by messages*.

For example, if you are driving a car you (an object) are sending messages to the car (another object) like "go faster," "go slower," "radio on," and so forth. The mechanisms that you use to convey the messages are pretty well standardised across cars: there's a "go faster" pedal, a "go slower" pedal, and a "radio on" button. Quite how the car works beyond this is kept well away from you (oil levels, petrol flows, etc.) and kept as *hidden internal state*, or *encapsuated*. And different cars may implement the operations in quite different ways, but you can still drive them, as they present a uniform *interface*.

> Aside: And, you, as an object, have an interface often pretty consistent with other people, sharing common language; and you have a large amount of hidden internal state in you head. The analogy is weaker here, though, as people do not react consistently to a given message!

This is the OO view of programming:

- the program consists of objects
- the objects invoke operations in other objects by sending messages amongst themselves
- the operations will generally send further messages
- the messages an object can process are given as a well-known interface
- the messages cause changes in the objects' internal state
- the internal state of an object is generally inaccessible to another object

The use of hidden state and published interfaces means that the object can be removed and replaced by a different implementation, and the user is non the wiser. This is *loose coupling*.

To make all this practical, other concepts are introduced. Rather than describe the behaviour and possible states of every object individually, objects are grouped into *classes* that share common behaviour. So a class describes

- *attributes*
- *operations*

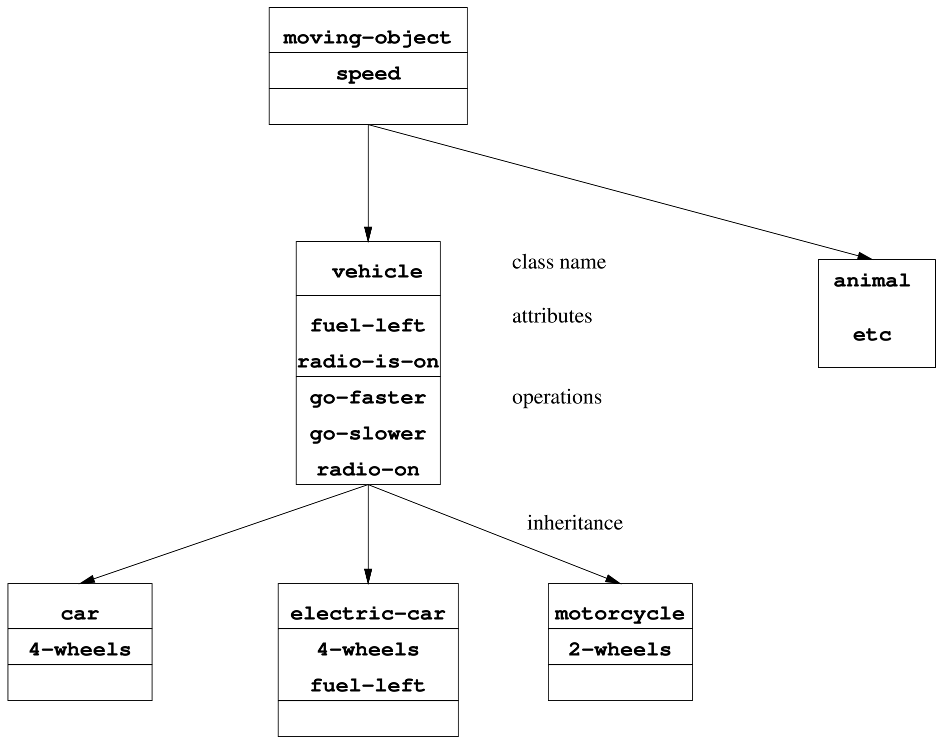


Figure 1: A Simple Class Hierarchy

of a bunch of objects. The class of `cars` would describe attributes, such as speed, amount of fuel left, and whether the radio is on; and operations such as accelerator and brake pedals and radio buttons. A particular car is an *instance* of the class.

As we generate classes, is it found that we can arrange them into hierarchies. Thus the classes of `cars` and `motorcycles` share certain attributes and operations like speed, fuel left. Other attributes are not shared, like number of wheels. Again to avoid replication, we can generate a new class named `vehicle` that contains just the attributes and operations common to both.

Notice that we can't make an instance of `vehicle`, as it leaves certain parts undefined, like the number of wheels. This is called an *abstract class*, and its purpose in life is simply to simplify our description process.

Now, the class `car` can be defined as a `vehicle` with certain extra features (like wheel count). We call `car` a *subclass* of `vehicle`. Ditto `motorcycle`.

> Aside: Note that through common usage we often say a car is an instance of `vehicle`. However, it is not a *direct instance*.

Now this is where OO is supposed to be a big win. We can define behaviour for both `car` and `motorcycle` by simply specifying behaviour for `vehicle`, and `car` and `motorcycle` *inherit* that behaviour. This abstraction of behaviour is the driving force behind OO programming.

We need not stop here. We can view `vehicle` as a subclass of a larger class, e.g., `moving-object`, and so on. We do need to make a decision on whether this proliferation of classes is actually helping: there are diminishing returns from all this!

Of course, the world is never that simple, and we find that things do not always fall into such neat hierarchies. For example, in an electric car the standard "fuel left" attribute does not make sense. Does that mean it can't be a subclass of `vehicle` and we have to rejig our nice picture?

The OO solution is to allow *overriding* of behaviour and attributes in subclasses. We can override the fuel left attribute in the `electric-car` to make it return something more suited to the purpose. In general, when looking for a behaviour or attribute we walk *up* the inheritance tree from our class until we find what we need. This ensures we get the most specific version.

Note the difference between OO programming and procedural programming:

- in OO only the operations defined on a class may access and modify the attributes of an instance, which has its own internal state
- inheritance gives us a natural way to reuse code
- in procedural it is hard to stop all and sundry changing the internal state of objects (e.g., structures) in any way they feel
- no inheritance

To summarise:

- everything is an object
- object pass messages between themselves
- each object has its own internal state (memory)
- every object is an instance of a class
- the class defines the behaviours for its instances
- classes are arranged in an inheritance hierarchy

> Aside: To whet your appetite for stuff later in the course, in some languages classes are also objects, and thus instances of other classes!

### 2.3 A Family Tree

You may have thought that concepts encapsulation and inheritance were enough to make all OOLs look alike. However, there is still plenty of room for variation.

Notice that this is a class hierarchy tree! The dotted lines are not inheritance but *instance links*.

> Aside: Many diagrams of inheritance trees show both inheritance links and instance links: this is a often source of confusion.

## 3 Using Class Centred and Object Centred Languages

We are now going to start to fill in meanings for some of the words we have seen. We start by comparing class centred and object centred languages. We shall do this by examining sample implementations of

(a) complex numbers: this is fairly simple, so we can concentrate on the ideas

(b) sequences: these are data structures that allow us to do iterations

We shall use C++ and Lisp as examples of class centred languages, and Javascript as an object centred language.

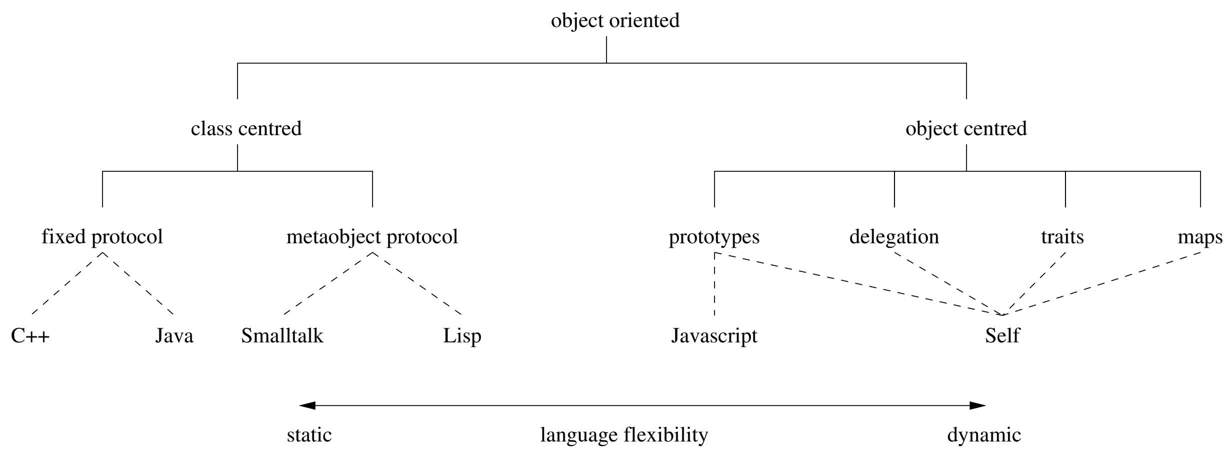


Figure 2: A Family Tree

## 3.1 Class Centred

Class centred languages are the most popular of the OO family, but this doesn't mean they are easy to use well. This is because the design process focuses on identifying abstractions—and these become classes in the implementation—rather than the actual objects themselves. This can lead to a over-generalised system not best suited to the problem at hand.

Class centred languages can be subdivided into two subgroups.

### 3.1.1 Class and Methods

In this subgroup

- methods are stored in (or associated with) the class (so a class knows all the methods that might be invoked on an instance)
- instance variables are *described* in the class (e.g., number of internal variables and types)
- instance variables are stored in the instances

Think of `structs` in C. The instance variables (elements of the struct) are described in one place (the `struct` definition), but each instance contains its own memory to store the variables.

Java and C++ are the major proponents of this subgroup, but there are many others like Smalltalk and Eiffel.

The methods in the class are the only pieces of code that can manipulate the slots in the instances.

### 3.1.2 Class and Generic Functions

In this subgroup

- methods are stored separately in *generic functions*. A generic function is a separate function-like thing that is not tied to any specific class. In other words a generic function can be defined to apply to instances of many different classes (and do different things on them).

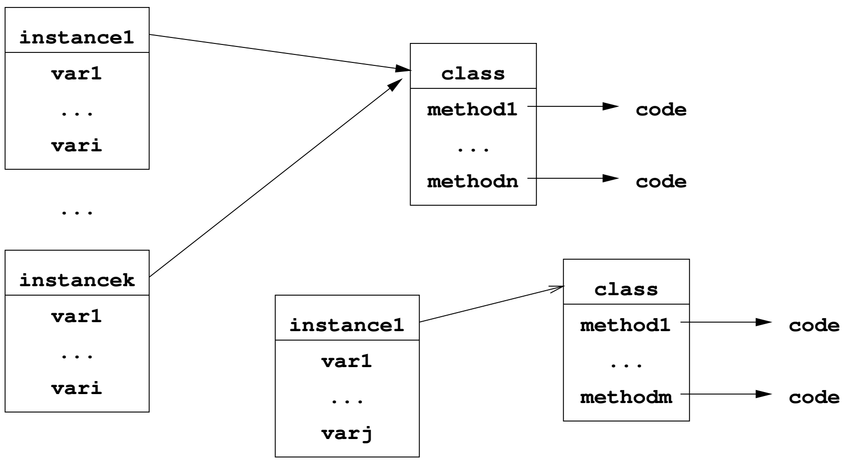


Figure 3: Class and Methods

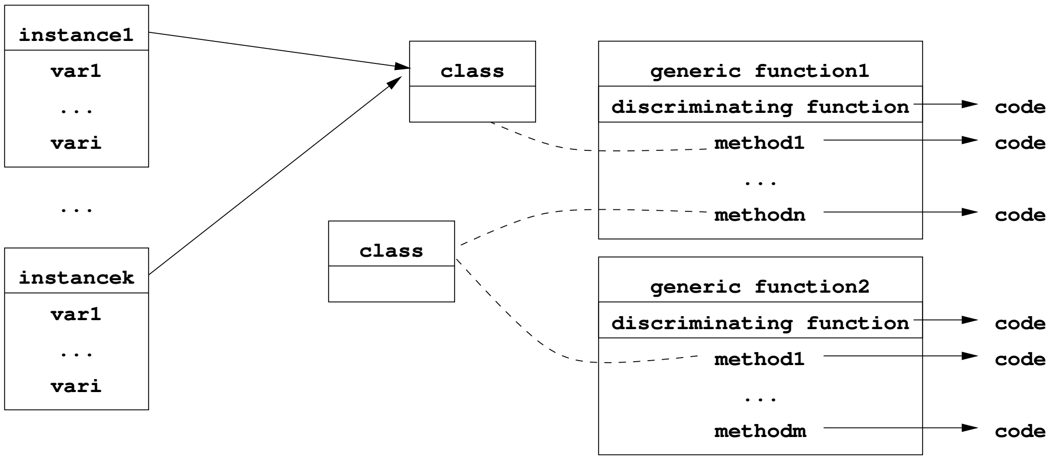


Figure 4: Class and Generic Functions

- instance variables are as in the class and methods subgroup.

The main proponent of this subgroup is Lisp.

Generic functions require a particular piece of code, the *discriminating function* that determines which of the methods it holds is appropriate to the instance(s) it is applying to.

Thus:

- class and method: classes hold the functions
- class and generic function: functions "hold" the classes

This difference is not just organisational: it also affects what the language can express.

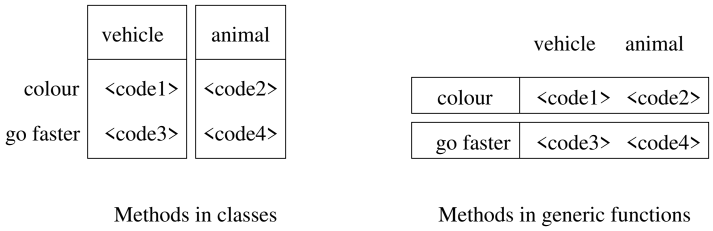


Figure 5: Methods vs. Generic Functions

Class and method

Send message to object. In C++

```
polynomial.degree()
node.degree()
```

the objects `polynomial` and `node` (of a graph) are sent the message `degree()`. In Smalltalk this is

```
^polynomial degree
```

The `^` is the *send* operator. With arguments this becomes

```
list.push(item)
^list push: item
```

This is called *single dispatch* as the choice of method is determined by *one* object (the receiver).

Class and generic function

Call the gf with object(s) as arguments:

```
(degree polynomial)
(degree node)
(push list item)
```

This looks identical to normal function call. It is called *multiple dispatch* as the choice of method is determined (via the discriminating function) by *all* the argument objects.

The syntax for class and method does not allow multiple dispatch, though the effect can be obtained through repeated use of single dispatch.

> Aside: Additionally, there are other syntax problems in C++. For example `1.max(7)` is not legal.

Both subgroups, though, share the basic OO principles: a client/application makes requests (sends messages to) a server/object, which makes appropriate responses. All the client knows is the interface, no more. The interface specifies *what* the object does, but not *how*.

## 3.2 Complex Numbers

We shall now implement complex numbers in an OO fashion. We shall do this in two stages:

1. write the specification (interface)
2. write the implementation

### 3.2.1 Complex Arithmetic

There are two common representations of complex numbers

1. cartesian: $x + iy$

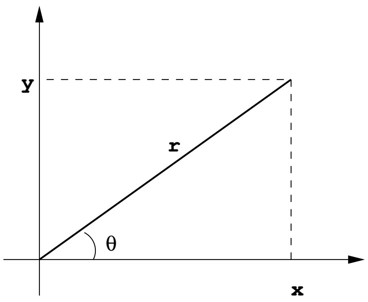


Figure 6: Cartesian and Polar

2. polar: $re^{i\theta}$

Our objective is to define arithmetic operations for both representations, and conversion from one to the other.

If $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$, then

$$\begin{aligned} z_1 + z_2 &= (x_1 + x_2) + i(y_1 + y_2) \\ z_1 - z_2 &= (x_1 - x_2) + i(y_1 - y_2) \\ z_1 z^2 &= (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1) \\ z_1 / z_2 &= z_1 \overline{z_2} / z_2 \overline{z_2} = z_1 \overline{z_2} / |z_2|^2 \\ &\text{where } \overline{z} = x - iy \end{aligned}$$

And for polar, if $z_1 = r_1 e^{i\theta_1}$ and $z_2 = r_2 e^{i\theta_2}$, then

$$\begin{aligned} z_1 + z_2 &= \text{convert to cartesian and add} \\ z_1 - z_2 &= \text{convert to cartesian and subtract} \\ z_1 z_2 &= (r_1 r_2) e^{i(\theta_1 + \theta_2)} \\ z_1 / z_2 &= (r_1 / r_2) e^{i(\theta_1 - \theta_2)} \\ \overline{z} &= re^{-\theta} \end{aligned}$$

Converting between the two:

$$\begin{array}{ll} x = r\cos\theta & r = \sqrt{x^2 + y^2} \\ y = r\sin\theta & \theta = \arctan(y/x) \end{array}$$

with all sorts of provisos about division by zero and choice of quadrants for arctan.

### 3.2.2 Design

OO design normally has three stages:

1. classes: what kinds of objects do we want
2. interfaces: what operations should they have
3. attributes: what information to store in the object

For our case the interface will be:

- arithmetic operators as above

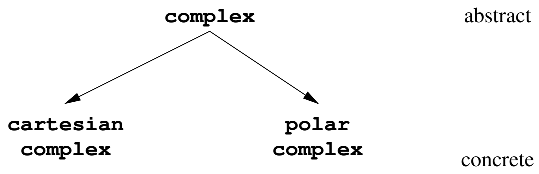


Figure 7: Complex Number Hierarchy

- a constructor (make a complex number)
- a destructor (remove it once we are done)
- a printer (so we can see its value)

Two slots:

- one for $x$ and one for $y$
- OR one for $r$, and one for $\theta$

Classes: we have two representations that are structurally identical (two slots), but we want to keep them separate. On the other hand, we want both representations to be complex numbers so we can manipulate instances regardless. The solution is to use an abstract class.

#### 3.2.3 Complex Numbers in Lisp

See the handout for the Lisp code.

#### 3.2.4 Complex Numbers in C++

in contrast to Lisp, C++ is statically typed: types are checked by the compiler and there is no run-time type checking.

Furthermore, (non-virtual) method selection is also done at compile time; hence the compiler is constructing the class hierarchy and determining method dispatch at compile time. This is generally more efficient (though run-time dispatch can be optimised quite well), but is less flexible (the static nature means no run-time creation of classes).

See the handout for the C++ code.

### 3.3 Object Centred Example

We now look at the same problem using an object centred language.

#### 3.3.1 Complex Numbers in Javascript

Javascript, like lisp, is dynamically typed. However, it is quite different from both Lisp and C++ in being a *prototyping language*. This means there are no classes, only objects.

There are just four kinds of things in JS:

1. numbers
2. strings
3. functions
4. objects

   Aside: A not-very-pure approach to OO in that it treats these things differently, not all uniformly as objects. Probably for efficiency reasons.

The prototyping style is to define an initialisation function and then call this with a fresh object, e.g.,

```
function cartesian_complex(x, y) {
  this.real = x;                       // instance variables
  this.real = y;
  this.add = cartesian_complex_add;    // method on +
  ...
}

...
z = new cartesian_complex(4,2);
```

The keyword `this` refers to the current object. The `new` creates a new object, while `cartesian_complex` initialises it.

Encapsulation derives from the data and methods being packaged into a single entity (in the initialisation function).

A limited kind of inheritance derives from creation time sharing of methods with other objects initialised by the same initialisation function. Thereafter the objects are independent.

See the handout for the JS code.

## 3.4 Sequences

As a second example, we shall consider sequences. This will illustrate

- inheritance
- genericity
- polymorphism

Our objective is to identify commons aspects of the behaviour of sequence datatypes (e.g., lists and vectors). Then we will be able to write *polymorphic* programs, i.e., programs that operate in terms of sequences in general, rather than relying on a particular representation.

Furthermore, the architecture should support the integration of new sequence types such that existing programs continue to work with the new types.

### 3.4.1 Requirements

What interface do we want? We could do loads, but to keep it simple:

- size: how big is the sequence?
- empty?: it is an empty sequence?
- insertion: add an element
- iteration: apply some function to each element in the sequence in turn

### 3.4.2 Design Sketches

We want the same operation to work for a range of datatypes. There are two styles of solution.

1. Generics. Implement an iteration (etc.) function for each sequence datatype. This does not scale well as the cost of adding a new datatype or a new operation is high.
2. Polymorphic. We should be able to get away with just *one* implementation of each algorithm. BUT how can sequences be manipulated regardless of their underlying structure?

   We can do this by introducing *iterators* and *iteration states* to acts as an interface between iteration operations and sequence datatypes.

An iterator interface has several parts:

- construct operator
- advance iterator to next state
- access current value
- compare iteration states

For example, to print a sequence we can write a polymorphic operator that goes:

1. construct an iterator for the sequence
2. repeat {
3.     access and print current value
4.     if current state is last state, we are finished
5.     advance iterator to next state
6. }

The iterator hides the details from us, presenting a polymorphic solution.

### 3.4.3 Sequence in C++

Again there are (at least) two approaches:

We can write the code ourselves.

+ we can keep it simple
+ we can make it small
− it will be limited
− it ignores re-use

We can use the C++ *Standard Template Library* (STL).

+ this is very sophisticated
+ it is an industry standard (part of ANSI/ISO C++ Standard, 100 pages)
− it is very complicated (on the inside)
− it is large

The STL contains five kinds of components:

1. containers: sequences (vector, deque, list), and associative (set, multiset, map, multimap, hashtable)
2. iterators
3. algorithms
4. function objects
5. allocators

We shall only look the first three.

Our plan of attack on the STL:

- a simple application
- containers
- iterators
- iterator adaptors
- algorithms

And then a look at iterator implementation, including references and templates.

### 3.4.4 STL Example, Version 1

We shall write a program to sort integers. It will

1. read integers
2. sort them
3. print them

Version 1 is basically standard C using the `qsort` library function (excepting the input/output). The construct

```
while (cin >> array[n++]);
```

reads integers from the iostream `stdin`, and stores them in successive slots in the array.

See the handout.

### 3.4.5 STL Example, Version 2

We now introduce the STL.

Some notes:

- includes `<algo.h>` for `sort`
- includes `<vector.h>` for the vector container class and vector iterators
- in `vector<int>` the `vector` is a *template class* while `int` is a *template parameter*. This is an example of a *parameterised type*. In general, `vector<type>` declares a vector that contains objects of type `type`
- A `vector<int>` can grow dynamically (it doubles in length when size reaches capacity)
- `v.size()` the number of elements currently stored
- `v.capacity()` the number of elements that could be stored
- `v[i]` the $i$th element of `v`. N.B., `v` is not a C array, and so this is an overloading of `[]`
- `v.push_back` appends an item to the vector, expanding it if necessary
- `sort` is written in terms of iterators, and so is independent of the data representation
- the sequence to sort is defined by two iterators: `v.begin()` for the start, and for the end `v.end()` (actually "the end plus 1," as is usual for C)
- iterator operators are:
  - `++` move forward one item
  - `*` current element (e.g., `*v`) c.f., `[]` above
  - `==` `!=` compare iterators

  we could write

  ```
  for (i = v.begin(); i != v.end(); i++) { ... *i ... }
  ```

Note that, although the program is shorter, that doesn't necessarily mean it is faster or more efficient!

### 3.4.6 STL Example, Version 3

Next we look at *iterator adaptors*. These are a way of making new interfaces for old programs. Our objectives are to make an input/output stream look like a sequence by providing a suitable iterator interface. And then to make a vector look like an expanding sequence by making `*i` use `push_back`.

**Case 1** The input stream `cin` has the correct functionality to be an iterator, as it delivers a sequence of elements. On the other hand, it has an incorrect interface to be a sequence: it has no `++` or `*` operators.

The function `istream_iterator` adapts an input stream to look like an iterator.

The sequence function `copy` makes a copy of a specified subsequence of one sequence to a given starting point in another sequence. Thus

```
copy(start, end, dest)
```

reads from `start` until `end` and stores the elements starting at `dest`.

We can do much the same with `cout`. The function `ostream_iterator` adapts an output stream to look like an iterator. The second argument to the constructor is the element separator.

**Case 2** The function `back_insert_iterator` adapts the iterator interface of the given sequence to

(a) ignore `++` operations

(b) use `push_back` instead of `*` on the lhs of a `=`

Observations:

- the final program is at a much higher level than the original
- operations are applied to whole containers rather than to the elements they contain
- STL algorithms work on ranges of elements, so we must have some means of specifying their beginnings and ends.

> Aside: Some people have difficulty with the idea of interfaces. Imagine a simplified interface to a car: a steering wheel, a go-faster pedal, and a go-slower pedal. This standard interface means anyone who can drive any car can drive *every* car. And if you fit a steering wheel and the two pedals to a spaceship, then you can drive a spaceship. This is *use of an interface to do the same kinds of things on different kinds of class.*
>
> More sophisticated is the concept of multiple interfaces to a single class. Above we have `back_insert_iterator` as an alternate interface to `vector`. Suppose you fixed a steering wheel to the carburettor of a car, a go-faster pedal to the headlamp switch, and a go-slower pedal to the horn. This is the *same* interface to an existing object: the old interface is still there. Turning the new wheel does something completely different to turning the wheel on the standard interface: it makes the car go faster or slower. Similarly the pedals do something different. It's the same car, just different ways of controlling it via the same interface. We can even use both interfaces simultaneously, as long as we are well aware of what's going on. e.g., the interaction between the new wheel and the old pedals. This is *use of an interface to do different kinds of things within the single interface.*

### 3.5 Templates

We saw the use of templates above. Now is a suitable time to look at them in more detail. C++ inherited its basic type language from C:

- an object's type is identified by a name (e.g., `int`, `struct foo`)
- a function's type is identified by a prototype (e.g., `int add(int,int)`)
- type matching is done by comparing type names
- this leads to type declarations which differ only in constituent types

E.g.,

```
struct int_list {
  int i;
  struct int_list *next;
};
```

```
struct dbl_list {
  double i;
  struct dbl_list *next;
};
```

These two list structures are "really" the same, just that one contains integers, the other floats. Many of the functions we will write to manipulate them will be identical: e.g., append to list, delete from list, length of list. It seems pointless to repeat code, it would be simpler to define a type like

```
struct T_list {
T i;
struct T_list *next;
};
```

and define our functions in terms of `T_lists`. Later we can instantiate `T` as `int`, say, and we have `int_list`, and then instantiate `T` as `double`, say, and we have `double_list`.

Templates extend the type language of C++ by introducing *type variables* which can later be replaced by actual types. Templates can be attached to

- classes
- functions
- structs

This allows us polymorphism in C++.

> Aside: The reality is that given a parameterised type the compiler effectively writes out the code multiple times with the parameter instantiated for each type we asked for, and then compiles that. There is no code sharing actually going on, so it's not real polymorphism.

### 3.5.1 Function Templates

We shall illustrate the use of function templates by writing a function that swaps the values of two variables.

It is simple to write a function that swap the values of two `int`s, or two `double`s, but impossible in C to write a general procedure (without generous use of casting and `void*`). Thus such a procedure must be implemented by each application that needs it instead of being written just once and supplied in a library.

To define templates, we prefix a statement with the `template` keyword, and replace specific types with the template type variable.

```
template <class T>
void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
...
int x, y;
...
swap(x, y);
```

Notes:

- `T` is not special, it can be any identifier
- the compiler instantiates a copy of the function for each type needed (not true polymorphism)
- we can use `inline` to avoid function calls
- the function is written once, but can be used in many contexts
- we can have `swap(int, int)` (`T` is instantiated as `int`), or `swap(double, double)`, (`T` is instantiated as `double`) but not `swap(int, double)` as there is no template to match the latter.

### 3.5.2 C++ References

The `&` in the above denotes a *reference*. This is like giving alternative names to objects. The form `X&` for some type `X` is read as "reference to `X`".

If we have

```
int i = 1;
int &r = i;
```

then `r` is an alternate name for the variable `i`. Doing `r++` is the same as doing `i++`. Wherever you see an `r`, just replace it by `i`. In particular this is used in function definitions, e.g.,

```
void inc(int& a) {
  a++;
}
...
inc(x);
```

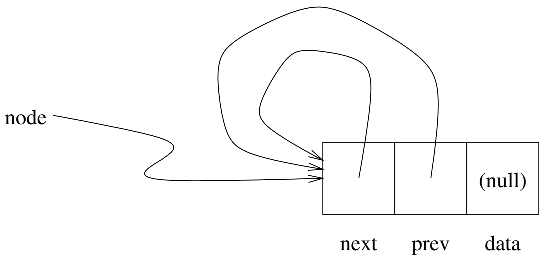


Figure 8: A New list Node

the local name `a` is an alias for `x`, so the function call is effectively doing `x++`.

This is often used in function call to provide updateable arguments, or for passing large structures or classes as arguments: passing a reference name is cheaper than passing the object (i.e., copying the object).

> Aside: References are implemented by means of pointers, but have the benefit of not requiring the programmer to remember where to put `&`s (address of) and `*`s.
>
> Exercise: what happens when we try to call `int(x+1)`?

#### 3.5.3 Class Templates

Our examples this time are minimal implementation of a vector type and a list type.

Notes on vector type:

- `private` data and functions only accessible to member functions in this class. E.g., `size` and `capacity` are not available outside this object, to ensure encapsulation
- constructor: a function with the same name as the class to make an instance of the class
- initialisers: expressions of the form

  ```
  data_member_name(expression)
  ```

  after the parameter list of the constructor. We could also do

  ```
  data_member_name = expression;
  ```

  within the body of the constructor, but the above is the only way to set a `const` data member.
- destructor: the class name plus a `~`, this clears up when we are done
- overloading `[]`
- iterators: `begin` and `end`
- side effects: what happens if `push_back` causes reallocation? The iterators are invalidated.

Our lists will be doubly linked, circular, with a distinguished initial/final node.

Notes on list type:

- NULL: there is no need for a null value because the beginning/end of the chain is identified by the known list node

- `push_front` and `push_back` both operations are $O(1)$ because of the circular and doubly linked technique
- simple: the special node avoids the need for special cases to handle the beginning and end of list
- iterators: we don't return the address of a node as this would break encapsulation. instead we construct a `list_iterator` object that encapsulates the state of the iterator
- destructor: more complicated than for the vector. We have to delete each node one-by-one while keeping a pointer to the next node. Also note that the root node is reinitialised for further use.

#### 3.5.4 Struct Templates

Pretty much as class templates, but less so.

### 3.6 Summary of STL Sequences

We have

- an introduction to STL as a user
  - containers
  - algorithms
  - iterators
  - iterator adaptors
- template declarations: extending the type language of C with type variables. We can write parameterised functions, struct and classes
- simplified implementations of C++'s vector and list classes to illustrate the use of templates
- the notion of iterator is the key to an open architecture for sequences. It abstracts away from the physical structure, making possible polymorphic algorithms, e.g.,

  ```
  for (i = s.begin(); i != s.end(); i++) {
    process(*i);
  }
  ```

  regardless of the types of the sequence `s` and its iterator `i`.

### 3.7 Sequences in Javascript

We shall reimplement the list type in Javascript, following the C++ example. The list object encapsulates a list node which marks the beginning/end of a circular list. Likewise, a list iterator object encapsulates a pointer to a node in the list and supports the operations of increment (`inc` or `++`), value (`val` or `*`), and not-equal (`neq` or `!=`).

Note that the `toString` method for list is very inefficient, but it was easy to implement. The `this` keyword, as in C++, refers to the current object. As in the C++ implementation, the `inc` operation returns the (modified) iterator object.

## 3.8 Sequences in (Eu)Lisp

And a look at a vector implementation in EuLisp. in contrast to C++ and Javascript, the operations are now grouped into generic functions.

Notes:

- `<extensible-vector>` EuLisp already has a `<vector>` class so we define our new class to inherit from that
- `initialize` the procedure to `make` a new object in EuLisp splits into two stages
  1. `allocate` allocate some memory. The method used to allocate an object is selected based on the class of the class of object
  2. `initialize` initialise the slots in the object. The method used to do this is selected based on the class of the object

  In fact, the `make` procedure is pretty much just

  ```
  (defun make (x)
     (initialize (allocate x)))
  ```

  So `(make <vector>)` calls `allocate`, a generic function that will choose a method based on the class of `<vector>` (that happens to be the class `<simple-class>` in some implementations of EuLisp). This allocates the right amount of memory for a vector. This chunk of memory is returned by `allocate` to `initialize` which then does its part. Notice `initialize` is also a generic function, and so chooses a method based on the class of the element returned by `allocate` (which will be a vector in this example).

  Compare this with

  ```
  new cartesian_complex(5,2)
  ```

  in Javascript. The `new` allocates a fresh empty object, and then the `cartesian_complex` initialises it.

  In EuLisp and Common Lisp the behaviours of `allocate` and `initialize` can be extended by the programmer. For example, we may wish to count the number of objects allocated; or allocate objects on disk, not in memory; etc.
- `call-next-method` will be explained later
- `(defmethod (setter element) ... )` to define a method to set a slot in an sequence. Here `element` is a generic function to access an element in a sequence, and `(setter element)` is a generic function to update an element in a sequence
- named `let` a way of defining a local function.

  ```
  (let label (initforms) body)
  ```

  Within the body the `label` refers to a function whose arguments are the variables of the `let` and whose body is the body of the let
- vector iterator: the representation encapsulates the vector and and integer index. Constructors as as in C++, operations `!` for indirection, and `!=` for comparison.
- destructor: none required in Lisp, as garbage is collected automatically
- the characters `.`, `!`, etc have no special meaning in EuLisp.

### 3.9 Summary of the Open Architecture

what are the benefits?

- separation of issues: by identifying iterators as the common vocabulary for working with sequences
- separation of sequence-dependent and sequence-independent issues: each class of sequence defines an iterator interface, but polymorphic algorithms are written for *all* classes.
- extensibility: new sequence type and new algorithms can both be integrated with little overhead
- balance between revealing enough of a mechanism to make it usable by both existing clients and yet-to-be invented clients and keeping enough hidden not to compromise integrity of efficient implementation.

This last is the key aspect for an open architecture designer and is an instance of a *design pattern*. Design patters are inspired by architecture and stem from the writings of Christopher Alexander in "Habitable Spaces."

## 4 Modelling Object Oriented Systems

Objective: to demonstrate how different kinds of OOL work.

What are the properties to model?

- Basic requirements: encapsulation, inheritance
- Organisational requirements: fixed vs. variable numbers of slots, creation vs. run-time sharing

We shall concentrate on three type of OOL. In increasing order of complexity of organisation:

1. prototyping: Javascript
2. delegation: Self
3. class: Java, C++, EuLisp, Common Lisp, Smalltalk, Eiffel

We shall analyse what makes an OOL work by looking at the simple problem of modelling a bank account and then generalising.

### 4.1 Classical Bank Accounts

See the handout of Lisp code. These are "classical" solutions (very much derived from Abelson and Sussman).

#### 4.1.1 Solution 1

A global variable `balance` models the attribute of an object; a global procedure `withdraw` models the operation of an object.

This has a few problems:

- the balance is not encapsulated
- we cannot create multiple accounts

| | |
|---|---|
| **Account** | name |
| **balance: int** | attributes (types) |
| **withdraw: int -> int** | operations (signatures) |

Figure 9: Classical Solution 1

### 4.1.2 Solution 2

We use a local (`let`) variable `balance` for attribute; a procedure `new-withdraw` for operation. The procedure is exported from the scope of the `balance` declaration.

Now the balance is encapsulated: the only way to refer to `balance` is by calling the function `new-withdraw`.

We still cannot create multiple accounts.

### 4.1.3 Solution 3

We expand on solution 2 by making a function that returns a new “withdraw” function. The local parameter `balance` is the attribute.

We can now create multiple accounts; each account is represented by a procedure comprising the code of `withdraw` and *separate* balances.

```
(setq acc1 (make-withdraw 100))
(setq acc2 (make-withdraw 200))

(acc1 10)        ; withdraw from account acc1
```

We can only withdraw: we want a means to deposit, too!

### 4.1.4 Solution 4

Now `make-account` returns the `dispatch` procedure. This maps a message (`withdraw:` or `deposit:`) to an operation.

We now have multiple accounts, and can deposit and withdraw. This solution is also easily extensible in that adding new methods is straightforward.

Each account is represented by a procedure comprising the code of `dispatch`, and has separate `balance`, `deposit`, and `withdraw`.

```
(setq acc1 (make-account 100))
(setq acc2 (make-account 200))

((acc1 deposit:) 20)   ; deposit into account acc1
((acc2 withdraw:) 30)  ; withdraw from account acc2
```

## 4.2 C Version

See handout of C code. Note this is not valid C++ as it uses the `this` keyword.

> Aside: A good way of annoying C++ programmers, particularly those who claim C++ is backwards compatible with C, is to use variable or functions named `new`, `delete`, `this`, and so on.

Notes:

- C does not have the environment capture properties of Lisp, so we have to pass the object about explicitly as an extra argument to a method, the `this` argument.
- The transition from C++ to C is essentially hiding the `this` argument: as an argument to a method (both in definition and call), and the `this->` within the body of the method.
- We use the `SEND` macro to tie the method to the object: `acc1->withdraw` gets the method for the object, and then we apply it to the object `acc1`. It is a good idea to make them the same object!

Notice how this naturally leads us to a OO view of the problem: messages and objects.

How well does our C sketch meet the original requirements?

+ encapsulation: we export operations only
+ inheritance: weak, as we have creation time sharing
+ fixed numbers of slots

This basic prototyping system is essentially equivalent to Javascript.

## 4.3 Conclusions

- objects have indefinite extent: we don't know how long an object is needed. This requires dynamic storage allocation (`malloc`/`free`)
- lexical scoping and procedural values were used to *simulate* encapsulation of attributes and operations. We can use structures to represent objects
- object operations need access to attributes and other operations on the object. Thus methods need to refer to the object (receiver of the message) in addition to method arguments. We can supply the object as the first argument to a method (C++, for example does this, but hides the fact from us)

# 5 Prototyping

The motto is:

> one kind of object, and no links.

The bank account example in Lisp and C demonstrates some of the essential properties involved in a simple object system:

- encapsulation
- multiple instances
- multiple operations (messages)
- storage allocation for attributes and operations (methods): these are kept in *slots* in the class or object or generic function
- a static slot declaration mechanism.

We also want inheritance, but we haven't got that far yet. We shall now look at ways of implementing the above, starting with the simplest kind of OO system, *prototypes*. Prototyping languages don't have classes or inheritance, so they are a good place to begin.

We have seen a prototyping language before: Javascript. The way to define a type is to show how to make an instance.

We want to add flexibility: the ability to add attributes and add operations requires a more flexible representation for the object. One way could be the use of a hash table: the message is the table key, the slot value is the table value. With this, slot access is a hash table lookup. Thus new slots can be added by making a new entry in the hash table.

See Prototyping 1.

We want to strengthen inheritance: this is simple in principle, but tricky in practice. A first step in this direction is called *delegation*.

## 5.1 Delegation

The motto is:

> one kind of object, and one kind of link.

The significant difference between a prototype object and a delegation object is the presence of a *parent* link.

When an object cannot process a message itself, it is passed to the parent to handle.

See handout on delegation in Lisp.

From the above, delegation seems quite simple: but this is not so when when return to the static (C) style of language. This is because we need to inherit attributes but share methods.

Look at the problem from a storage allocation angle:

There are three categories of information:

1. object organisation fields: slots, parent, etc.
2. attributes of object
3. operations of object

Category 1 is for data integral to the system: what makes the system work. Categories 2 and 3 are for object data defined by the user: what make the application work.

To make this distinction explicit (for emphasis only) we shall represent 1 by one data structure, and 2 and 3 by another.

See handout Delegation in Practice.

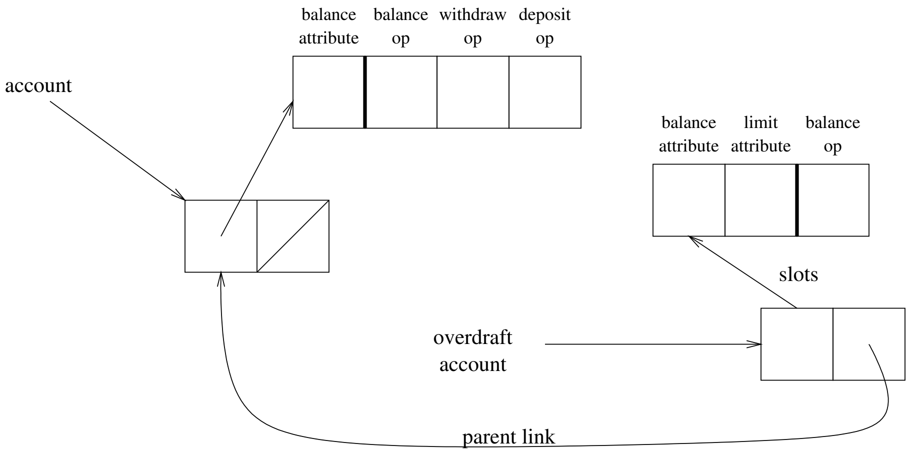


Figure 10: Delegation Storage

## 5.2 Strengthening Inheritance

We will need a more complicated method lookup.

With (static) prototyping there is a 1-to-1 mapping of messages to methods, so "send message" is actually structure access, and no computation is needed at run-time. Methods can be resolved at compile-time.

With delegation we need to search for the method. This can happen

(a) at compile-time in static language (e.g., C++)

(b) at run-time in dynamic languages (e.g., Smalltalk)

The process is equivalent to that in the Lisp example: search up the parent chain until

(a) we find the method, or

(b) we reach the root

Generally, case (b) is an error.

See delegation handout.

### 5.2.1 Delegation Method Lookup

How do we identify the correct method? A simple way is map each message to an integer $n$, such that any object will respond in the same way to the same message as any of its parents.

Deeper objects have more attributes, but we only store operations in the objects that define them. By keeping count of the numbers of attributes and operations defined in the current branch of the inheritance tree we can tell whether a message (i.e., an integer value) is defined in this object, or a parent. Also, to find the operation when it is defined in the current object, it is a simple matter of subtracting an offset from the integer value.

The size of an object is the number of attributes plus the number of methods newly defined on this object. Plus some bookkeeping like

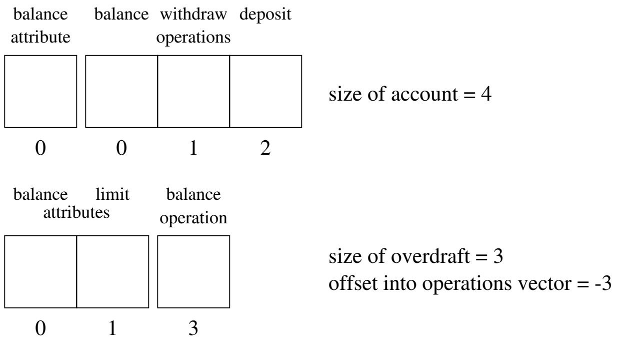


Figure 11: Delegation in Practice

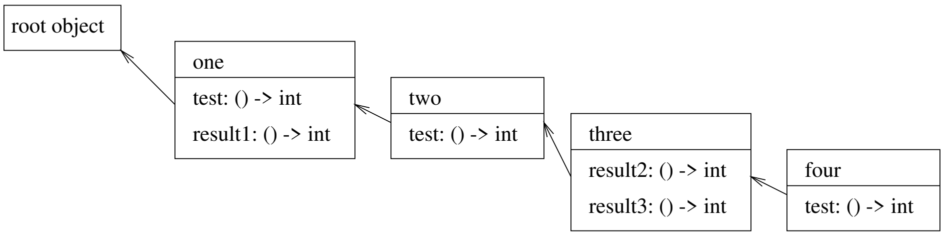


Figure 12: Smalltalk Test

- parent link
- total number of methods defined.

The latter is needed to compute the offset.

We also need a root object to terminate the upward search, and to store the default methods that apply to all objects.

See code for `overdraft.c`

There is a big hole in this technique: it fails to provide method overriding. The example cheats by using a different message to get the balance from `account` (`ac_balance_operation`) than from `overdraft account` (`od_balance_operation`).

To fix this requires another jump in complexity.

### 5.2.2 Delegation Method Lookup Again

The Smalltalk-80 book (Goldberg and Robinson) gives a useful artificial example to illustrate the correct treatment of method lookup in the presence of overriding.

This example defines a few classes with some methods, and calls those methods.

Methods:

- `test()` on an instance of

  `One` → 1

  `Two` → 2

  `Four` → 2

- `result1()` → result of `self.test()`
- `result2()` → result of `self.result1()`
- `result3()` → result of `parent(self).test()`

Suppose `e1` is an instance of class `one`, `e2` is an instance of `two`, and so on. We send the following messages:

- `e1.test()`
- `e1.result1()`
- `e2.test()`
- `e2.result1()`
- `e3.test()`
- `e4.result1()`
- `e3.result2()`
- `e4.result2()`
- `e3.result3()`

See handouts for solutions using Lisp-based delegation.

To illustrate the behaviour of objects and methods in the delegation test, consider each send of `test`

1. `(send one test:)` returns 1
2. `(send one result:)` calls `(send one test:)`, returns 1
3. `(send two test:)` returns 2
4. `(send two result1:)` finds `result:` in `one`, calls `(send two test:)` since `self` is `two`, returns 2
5. `(send three test:)` finds `test` in `two`, returns 2
6. `(send four result1:)` finds `result1:` in `one`, calls `(send four test:)`, returns 4
7. `(send three result2:)` calls `(send three result1:)` which it finds in `one`, so calls `(send three test:)` which it finds in `two`, returns 2
8. `(send four result2:)` finds `result2:` in `three`, then as previous with `four` in place of `three`, returns 4
9. `(send three result3:)` calls `(send two test:)` as `two` is `three`'s parent, returns 2.

The Lisp solution is straightforward as an object knows its own type: that is, an object's type is not determined by the variable it is held in, but is intrinsic to the object. Thus the type of `self` is never in doubt. On the other hand, in C the type of an object is determined the the type of the variable you hold it in. For example, a 32 bit chunk of memory is regarded as an `int` if you use a variable declared to be `int`; and it is a `float` if you use a variable declared to be `float`. The problem here is, say, a variable of type `vehicle` may contain an instance of `vehicle`, *or an instance of any subclass* of `vehicle`, say a `car`. Thus merely looking at the type of the variable, which is what C++ does, is not enough to determine the true type of the object.

So how can we handle overriding in the C implementation? A solution is to create a second category of method: *virtual methods*. These rely on the object knowing its own type somehow. These methods can be safely overridden, as we can always pick the correct type based on information in the object.

```
#include <stdio.h>

class vehicle {
public:
  int speed;
  vehicle() { speed = 0; }
#if 0
  virtual
#endif
    void inc(int x) { speed += x; }
};

class car: public vehicle {
public:
  int colour;
  void inc(int x) { speed -= x; }
};

int main()
{
  vehicle v;
  car c;
  vehicle *list[2];

  v.speed = 42;
  c.speed = 42;

  v.inc(3);
  c.inc(3);

  printf("v speed = %d\n", v.speed);
  printf("c speed = %d\n", c.speed);

  list[0] = &v;
  list[1] = &c; // a car is a vehicle

  list[0]->inc(3);
  list[1]->inc(3);

  printf("v speed = %d\n", v.speed);
  printf("c speed = %d\n", c.speed);

  return 0;
}

/*
No virtual:

v speed = 45
c speed = 39
v speed = 48
c speed = 42
```

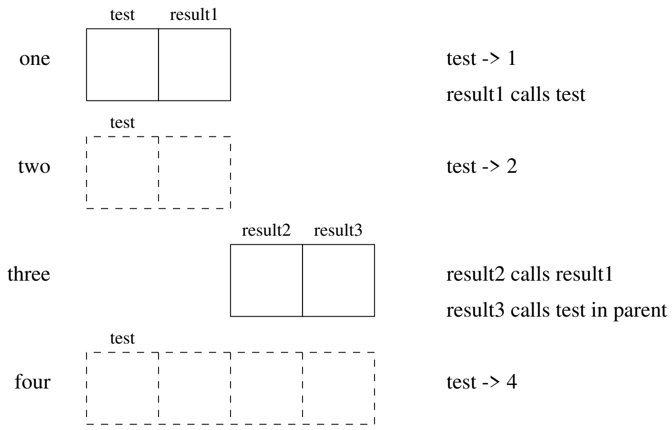


Figure 13: Delegation with Virtual Methods

```
Virtual method:
v speed = 45
c speed = 39
v speed = 48
c speed = 36
*/
```

Thus

- selection of a non-virtual method is done on the basis of the type of the variable (or expression) we are looking at (and can be done at compile time), while
- selection of a virtual method is done on the basis of the type of the object concerned (and is generally done at run time).

This allows us to do the delegation test in C (or C++).

This means no overhead for non-virtual methods, but the cost of calling a virtual method is somewhat larger. C++manages virtual methods by each instance containing a pointer to the appropriate virtual method. So the size of an object in C++ is dependent on the number of virtual methods its class has. The vehicle instances above will have a hidden slot that contains a pointer to the appropriate `inc` function: an instance of `car` will contain a pointer to the `inc` for `car`, while a direct instance of `vehicle` will contain a pointer to the `inc` for `vehicle`. Sending the message `inc` is compiled as "call the function contained in the `inc` slot."

Non-virtual methods are selected at compile time. Thus we have

- instances when there are virtual methods are bigger than instances when there are none, and the extra size is proportional to the number of virtual methods
- calling virtual methods is a little slower than non-virtual methods, as we have an extra indirection for every call.

See code for `one.c`

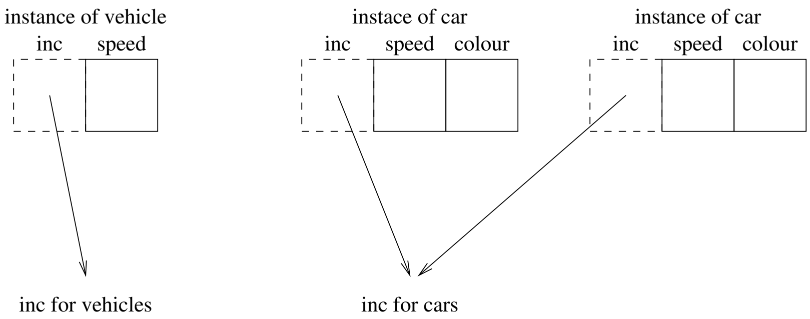


Figure 14: Virtual methods in C++

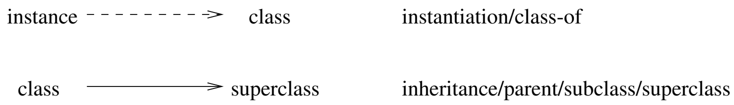


Figure 15: Instantiation vs. Inheritance Links

# 6 Classes

The motto is:

> two kinds of object, and two kinds of link

The two kinds of object are *classes* and *instances*, and the two kinds of link are *inheritance* (between classes) and *instantiation* (between a class and an instance).

> Aside: "inheritance" and "instantiation" move downwards. In the opposite direction the links are often called *parent* and *class-of*, respectively. Furthermore, inheritance links are also called *subclass* or *superclass* links.

> Aside: If you remember any set theory, you may like to think of classes as sets, and instances as elements in the sets. The inheritance relation is subset ($\subset$), and instantiation is membership ($\in$). This analogy can't be taken too far: see later.

Class based systems formalise the segregation illustrated in the last example.

| | |
|---|---|
| instances store attributes | classes store operations (and instance structure) |

When drawing diagrams containing classes and instances we shall need to distinguish the two type of link. A dotted line will denote *instantiation*, while a solid line will mean *inheritance*.

In delegation we found we needed a root object to inherit from: what happens with classes? Our next objective is to introduce the idea of a *reflective* object system, and to establish the initial class hierarchy of a class based system. A reflective object system is where the program can manipulate its own classes and class hierarchy. Not all (in fact not many) class based object systems are properly reflective: Lisp and Smalltalk are the main examples. A limited kind of reflection is available in other languages.

We shall need

- to define new classes: but a class, being an object in our system, will need to be an instance of something. Solution: a special class named `class` (say) whose instances are classes

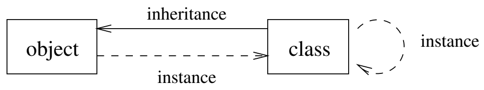


Figure 16: Initial Class Hierarchy

- attribute inheritance. Solution: an instance must contain *all* the attributes defined in this class and all of its superclasses (as with prototypes)
- operation inheritance. Solution: search for methods starting in the class of the instance, then moving up to the superclasses
- instance creation. Solution: send a `make` message to the class, which can respond appropriately. This appears as the construction function in C++, and the `make` function in Lisp.

## 6.1 Initial Class Hierarchy

Out initial reflective class hierarchy contains two objects:

1. the class `object`, the root of the inheritance hierarchy
2. the class `class`, the root of the instantiation hierarchy.

How are `object` and `class` related?

- `class` is the class of all objects (directly or indirectly via other classes). Thus `object` must be an instance of `class`
- `object` is the superclass of all classes (directly or indirectly) thus `class` must inherit from `object`
- of what is `class` an instance? Again, `class` is the class of all objects (directly or indirectly), so the class of `class` is `class`.

  Aside: This is where the set theory analogy fails: if `class` is an instance of itself, this would be $c \in c$ in terms of sets. Naive set theory (actually Zermelo-Fraenkel set theory) contains the Axiom of Foundation that states precisely the opposite of this, i.e., a set is never a member of itself. Thus Naive set theory cannot model classes in this way.

  Theories other than set theory have been developed that can cope with this self-membership.

In other words

- `class` is a subclass of `object`
- `class` inherits from `object`
- `object` is an instance of `class`
- `class` is an instance of itself
- class-of `object` is `class`
- class-of `class` is `class`
- etc.

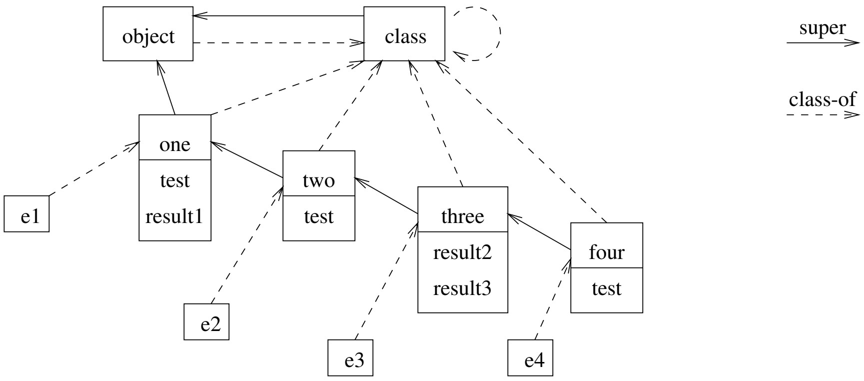


Figure 17: Test Hierarchy

### 6.2 The Smalltalk Method Lookup Test

In terms of the Smalltalk test, the hierarchy is a little more complicated.

We shall be dealing with reflective systems later.

### 6.3 Multiple Dispatch

To finish off this section we need to talk a little about the way methds are chosen when we send a message to an object.

All the method lookup operations so far have started from the idea of sending one message to an object. This is single dispatch.

But, for many operations the choice of method depends on more than one object, e.g., mixed-mode arithmetic. C++ resolves (or avoids) the issue through overloading the arithmetic operators, but the general case `x op y`, where `x` and `y` can have types `A` and `B` must be translated into

```
x.op(y)
```

with method definitions

```
void A::op(A y) {                void A::op(B y) {
  y.opAA(this);                    y.opAB(this);
}                                }

void B::op(A y) {                void B::op(B y) {
  y.opBA(this);                    y.opBB(this);
}                                }
```

Only now can address the problem of writing the code for the methods themselves.

This is rather tiresome, particularly when there are many types, or we want to dispatch on three or more arguments. Given $n$ types and $m$ arguments we are looking at an $O(n^m)$ problem.

See handout on multiple dispatch in C++.

Generic functions relieve the programmer of this burden at the cost of a more complicated (and costly) lookup algorithm.

See handout on multiple dispatch in Lisp, and the method determination algorithm.

```
(defclass A () ())
(defclass B (A) ())

(defgeneric op (x y))

(defmethod op ((x A) (y A)) "one")
(defmethod op ((x A) (y B)) "two")
(defmethod op ((x B) (y B)) "three")

(setq a (make A))
(setq b (make B))

(op a a) -> one
(op a b) -> two
(op b a) -> one
(op b b) -> three
```

- `(op a a)`
  - Applicable methods: `one`
  - Sorted methods: `one`
- `(op a b)`
  - Applicable methods: `one`, `two`
  - Sorted methods: `two`, `one`
- `(op b a)`
  - Applicable methods: `one`
  - Sorted methods: `one`
- `(op b b)`
  - Applicable methods: `one`, `two`, `three`
  - Sorted methods: `three`, `two`, `one`

Another example. The same classes as before, but now the methods are

```
(defmethod op ((x A) (y B)) "four")
(defmethod op ((x B) (y A)) "five")
(defmethod op ((x B) (y B)) "six")
```

- `(op a a)`
  - Applicable methods: none
  - Sorted methods: none. The system signals an error.
- `(op a b)`
  - Applicable methods: `four`

  - Sorted methods: `four`

- `(op b a)`
  - Applicable methods: `five`
  - Sorted methods: `five`

- `(op b b)`
  - Applicable methods: `four`, `five`, `six`
  - Sorted methods: `six`, `five`, `four`

In the last example `five` is sorted before `four` since the leftmost argument `b` is more closely matched with `B` in `five` than the `A` in `four`.

Notice that with a single argument, this algorithm reduces to the expected behaviour of picking the closest defined method.

Although this seems very expensive, the algorithms is only a *specification.* Actual costs can be reduced by

- on-the-fly recompilation: e.g., Self
- method caches: e.g., Lisp

As with all aspects of OO systems: it is easy to describe and implement the principles of an OOL; but it is hard to make it run fast.

# 7 Review

We have explored the implementation of three object models of increasing sophistication.

1. Prototyping. "One kind of object, no links"
   - encapsulation
   - creation-time inheritance
   - find-method protocol: does object have an operation matching the message?
   - examples: Javascript
2. Delegation. "One kind of object, one kind of link"
   - augments prototype with a *parent* link
   - encapsulation
   - attribute inheritance at creation time
   - operation inheritance at run time
   - find-method protocol: if object does not have an operation matching the message, delegate to parent
   - examples: Self
3. Classes. "Two kinds of object, two kinds of link"
   - segregates attributes from operations: instances store attributes, classes store operations
   - encapsulation

- attribute inheritance at creation time
- operation inheritance at run time
- find-method protocol: if class of instance does not have an operation matching the message, look it up in the superclass
- examples: Smalltalk, C++
- multiple dispatch find-method
- examples: Common Lisp, EuLisp

At each stage there is an increase in

(a) the overheads of the OO system

(b) the costs of trying to analyse away the overheads

This illustrates the tradeoff between the efficiency of static techniques, and the more costly dynamic techniques:

| static | $\longleftrightarrow$ | dynamic |
|---|---|---|
| simple representation and allocation | attributes | aids evolutionary |
| can compile away method lookup | operations | development and |
| can compile away inheritance overheads | inheritance | adaptive solutions |

We shall see later how the use of the meta level can increase flexibility at minimal cost.

# 8 Multiple Inheritance

We now consider some more advanced aspects of OO systems.

## 8.1 Multiple Inheritance

Multiple Inheritance (MI) is based on the observation that we might want to have objects that inherit behaviour from more than one parent. Thus, for example, an amphibious vehicle might inherit from both vehicles and boats. Or an `iostream` inherits from both an `istream` and an `ostream`. However, things are not as simple as we might hope.

So far we have used *single inheritance* (SI), where a class is a direct subclass of exactly one other class. The generalisation that MI gives us is to allow classes to inherit from more than one parent.

In SI, we get an inheritance tree: there is a unique path from a class to the root. With MI, we have a *directed acyclic graph* (DAG). Sometimes this is called a class *heterarchy*.

There are three objectives to this section:

1. to talk about algorithms for computing the *class precedence list* (CPL), which determines which attribute or operation to inherit if more than one parent offers it
2. discussion of MI in C++
3. discussion of concerns about MI

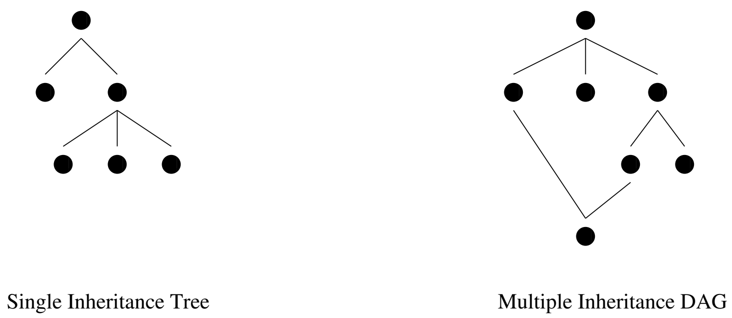


Figure 18: SI and MI

Why should we use MI?

- it may offer a better fit with the situation being modelled
- it may offer a better sharing or re-use of code

However, there are drawbacks:

- we will have to understand how to use complex MI organisations
- we will have to deal with the unpredictability (non-intuitivity) of CPL algorithms as we add more classes

These make MI somewhat hard to use effectively in practice. That this is true can be seen from the fact that the number of methods required in a class-and-method system increases as the $n^{\text{th}}$ power of the number of classes involved when dispatching on $n$ arguments.

Using generic functions relieves the burden somewhat, but this is at the cost of a much more complicated method lookup algorithm.

Recall the handouts for Lisp multiple dispatch method determination algorithm.

## 8.2 The Class Precedence List

Suppose we want to make an amphibious vehicle class by MI from a vehicle class and a boat class. The problems start when we get an attribute (or operation) from both parents.

In the diagram `amphib` inherits an attribute `colour` from both its parents `vehicle` and `boat`. However, it is the `same` attribute, as both of the parents inherited it from a common superclass, `moving object`. There is no problem here.

On the other hand, both `vehicle` and `boat` define an attribute named `speed`. Presumably, `vehicle` measures speed in MPH, while `boat` measures it in knots. There is a common purpose to both the attributes here, but in general there need not be any relationship between attributes of the same name in different classes.

So what `speed` attribute does `amphib` inherit? The answer is provided by the *class precedence list*.

In SI we don't have this problem: an attribute comes from a single direct or indirect parent. If the attribute is overridden in a subclass, we take the nearest definition. The CPL reduces the multiple-superclass DAG to a single list—a *linearisation* of the DAG. Then we can follow the example of SI, and pick the nearest definition. For example, we might linearise the superclasses of `amphib` as

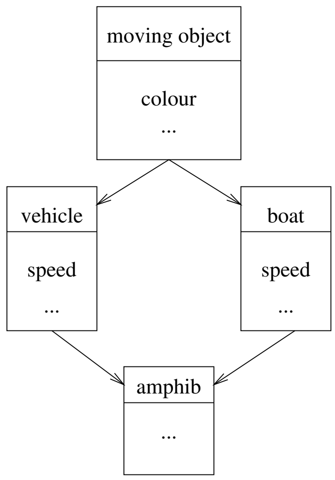


Figure 19: Multiple Attributes

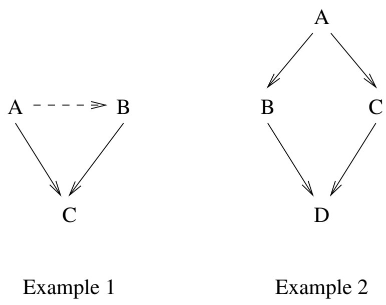


Figure 20: Example of CPL

`amphib`, `vehicle`, `boat`, `moving object`,

or equally as

`amphib`, `boat`, `vehicle`, `moving object`.

A CPL algorithm does this for us: it takes the DAG of superclasses living above our new class and produces a linearisation. The problem is to produce a CPL algorithm that is both self-consistent and give intuitively correct results.

### 8.2.1 CPL Examples

The idea is to determine what we want from a CPL algorithm by looking at a few example cases.

**Example 1.** C inherits from A and B. The tendency for left-to-right readers is to presume A comes before B. This is a left-to-right ordering.

**Example 2.** There at least 4 possibilities. Doing standard traversals of the graph above D:

- DBACA: depth-first, left-to-right
- DBCA: breadth-first, left-to-right

- DCBA: breadth-first, right-to-left
- DCABA: depth-first, right-to-left

Depth-first favours more distant classes over closer classes, unlike breadth-first. Breadth-first is usually preferred, as we usually want closer classes to override further ones. We also need to deal with duplicates. So we might reduce DBACA to DBAC or DBCA.

**The FLAVORS Algorithm.** FLAVORS was one of the first documented MI systems. This is a depth-first traversal in which the *first* occurrence of a duplicate is retained. Thus we get DBAC.

> Aside: FLAVORS as in ice-cream flavours. See mixins, later.

**The LOOPS Algorithm.** Developed at Xerox Parc. A depth-first traversal in which only the *last* occurrence of a duplicate is retained. Our example gives DBCA.

**The CLOS Algorithm.** The fundamental failure of FLAVORS and LOOPS is that they take no account of proximity, and it is easy to construct examples where deeper nodes are favoured over nearer ones, or the left-to-right precedence is ignored.

See handout for pathological inheritance examples.

The Common Lisp Object System (CLOS) algorithm is significantly more complicated, but gives better results. It still can get confused, though.

CLOS uses *local class precedence*, which uses the order of the superclasses as they are named in the class definition to favour nearer parts of the class hierarchy. It uses the rules:

- a class preceeds all its superclasses
- a class preceeds all of the classes listed after it in the superclasses list in a `defclass`

and produces a *topological sort*. This is just any linearisation that respects the above orders. If there is any ambiguity at any point about which class should be next, a tie-breaker is used. This says the class whose direct subclass occurs rightmost in the CPL constructed so far is to occur next. This tries to preserve simple chains of subclasses.

Example (middle graph on handout).

```
(defclass A () ())          A  B  C
(defclass B () ())          |\/  /
(defclass C () ())          |/\ /
(defclass D (A B) ())       D  E
(defclass E (A C) ())        \/
(defclass F (D E) ())         F
```

- FLAVORS gives the CPL at `F` to be `FDABEAC` which becomes `FDABEC`.
- LOOPS drops the first `A`, and becomes `FDBEAC`. Note that this breaks the left-to-right precedence for `A` and `B`.
- CLOS starts with the inequalities:

$$\begin{gathered} \mathtt{F} < \mathtt{D}, \mathtt{F} < \mathtt{E}, \mathtt{E} < \mathtt{A}, \\ \mathtt{E} < \mathtt{C}, \mathtt{D} < \mathtt{A}, \mathtt{D} < \mathtt{B}, \\ \mathtt{A} < \mathtt{B}, \mathtt{A} < \mathtt{C}, \mathtt{D} < \mathtt{E}, \end{gathered}$$

We easily get F < D < E < A < (C ? B), but we don't know which of B or C comes next. Apply the tie-breaker: B has direct subclass D, C has direct subclass E, and E is rightmost. So C comes next. The result is FDEACB.

Another example:

```
(defclass A () ())                   A
(defclass B (A) ())                 / \
(defclass C (A) ())                B   C
(defclass D (B) ())                |   |
(defclass E (C) ())                D   E
(defclass F (D E) ())               \ /
                                     F
```

What is the CPL for F? We get F then D or E? Local precedence says D next.

So far: F, D then B or E? We must use the tie-breaker. We see that B has subclass D which is to the right of subclass F of E. Thus B is next.

So far: F, D, B then A or E? From E < C and C < A we must have E next.

So far: F, D, B, E then A or C? Again, C is next.

The final CPL: F, D, B, E, C, A. We get B before E, though E feels as it it should be earlier.

The CLOS rules do mean that sometimes we can't create a CPL:

```
(defclass A () ())
(defclass B () ())
(defclass C (A B) ())
(defclass D (B A) ())
```

The CPL at D is D, B, A. The CPL at C is C, A, B. So far, so good. If we now have

```
(defclass E (C D) ())
```

we must have A < B and B < A. CLOS will signal an error in this case.

In the last example on the handout we insert a new class X between D and A.

- FLAVORS: FDXABEAC, becoming FDXABEC.
- LOOPS: FDXABEAC, becoming FDXBEAC
- CLOS: FDXBEAC.

A curiosity of the CPL for CLOS (and LOOPS) is the following: if each of A, B and C define a attribute P, then F inherits the P from B. The CPL at D is DXAB so D inherits P from A; the CPL at E is EAC so E inherits P from A. Thus, while both direct superclasses D and E of F inherit P from A, F itself inherits P from B. This is counter intuitive, and is a failing of the CLOS algorithm.

In conclusion:

- simple CPL algorithms only work for simple graphs

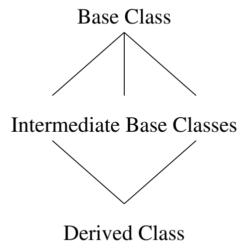


Figure 21: Terminology for MI in C++

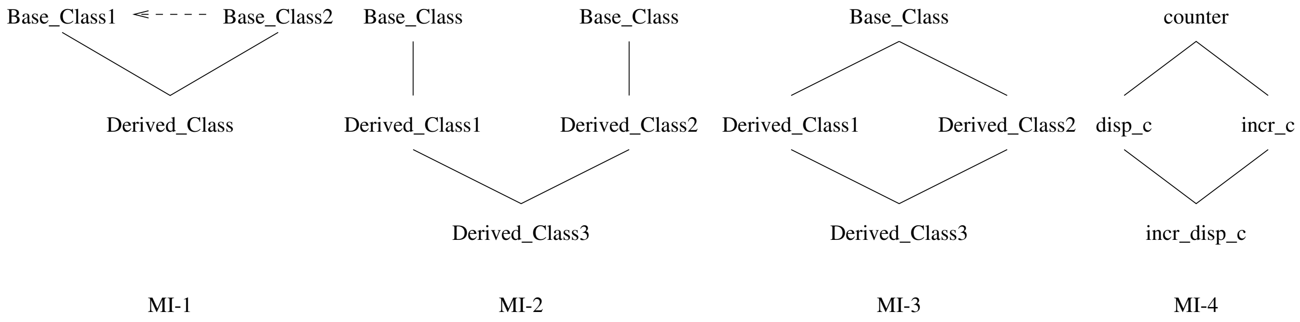


Figure 22: Examples of MI in C++

- complex graphs can lead to quite unexpected behaviours, and non-local consequences of local changes (c.f., the third example)
- MI is powerful, but also very dangerous. Use with care!

In order to predict the behaviour of a class we need to know

- the complete class hierarchy, not just the immediate parents,
- the linearisation algorithm.

## 8.3 MI in C++

There is no CPL algorithm in C++, it's all up to the programmer.

This is bad, since it makes the programmer do extra work; equally it is good since it doesn't get it wrong. The programmer must either

(a) control base class replication through virtual inheritance, or

(b) explicitly designate the origin of a member.

See handouts.

**Example 1.**

- no member name clashes
- instance order specified by the declaration of Derived_Class

In

```
class Derived: public Base1, public Base2 {
...
};
```

the CPL (as it were) for `Derived` is `Derived`, `Base2`, `Base1`. The constructors for `Derived` are called in the order `Base1`, `Base2` then `Derived`. This is depth-first left-to-right traversal of the DAG above `Derived`.

**Example 2.**

- if `Base_Class` is not inherited virtually, each intermediate base class gets its own copy
- thus the member `a` is ambiguous unless it is fully qualified, e.g.,

  ```
  Derived_Class::a
  ```

- Base_Class constructor is called twice

This is depth-first left-to-right traversal of the DAG, including repetitions.

**Example 3.**

- virtual inheritance ensures there is only one *shared* copy of `Base_Class`.
- `Base_Class` constructor is called just once

This is depth-first left-to-right traversal of the DAG, only including the first of any repetitions.

**Example 4.**

- this is no different in principle from MI-2
- it attempts to demonstrate a trivial application of MI
- note the call to the base class constructor from the initialisers of the other classes
- for a real example of MI see `/opt/gnu/include/g++/iostream.h`

Aside: We can make the inheritance virtual or not virtual. What effect does this have on the constructors?

```
class B { };
class D1: ?virtual? public B { };
class D2: ?virtual? public B { };
class D3: public D1, public D2 { };
```

| D1 | D2 | make D3 |
|---|---|---|
| not | not | B, D1, B, D2, D3 |
| not | virtual | B, B, D1, D2, D3 |
| virtual | not | B, D1, B, D2, D3 |
| virtual | virtual | B, D1, D2, D3 |

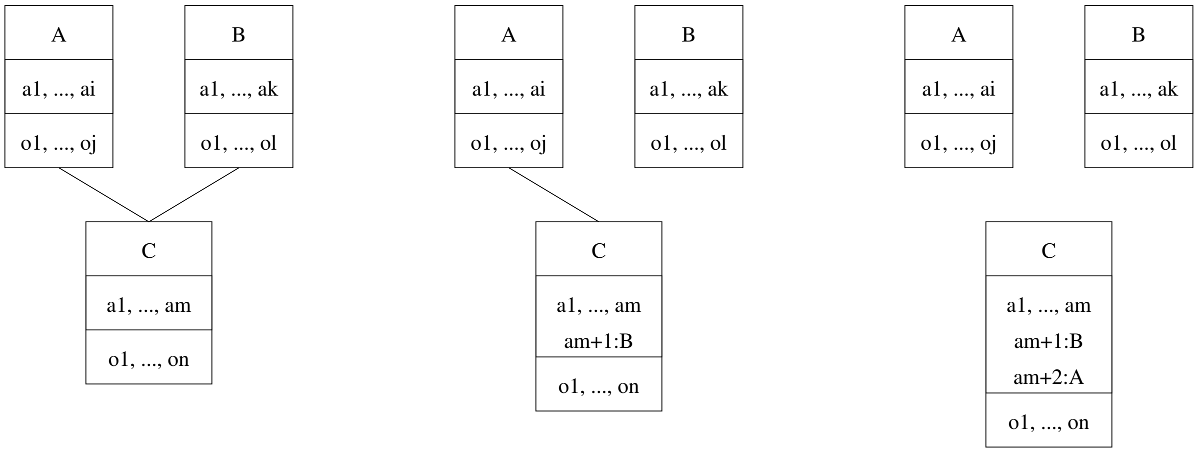


Figure 23: MI by Embedding

## 8.4 The MI Debate

Opinion is divided on the use of MI.

Those against MI examine examples of its use and usually demonstrate the class hierarchy can be rearranged to avoid use of MI. For example, *embedding* (also known as *composition*).

This introduces an additional attribute of type B into C. The new attribute contains all the slots in B. It provides "inheritance" of attributes, but not of operations. The argument can be taken further to dispense with the inheritance from A, too.

Thus

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

becomes

```
class A: public B {
  C instanceC;
  ...
};
```

plus some accessors to make using the `C` attributes not too tedious, or even

```
class A {
  B instanceB;
  C instanceC;
  ...
};
```

But then we are avoiding using any of the inheritance that C++ can provide.

Those in favour of MI argue that it

- can model problems better, e.g., iostreams
- it leads to better reuse and sharing of code, e.g., iostreams.

In general, since MI makes the class hierarchy more complex, caution is advised. See

- Somerville
- Booch
- Gamma et. al.

### 8.5 Summary of MI Problems

- Name collision. A syntactic issue: the derived class must fix it (the fault is theirs, not the various parent classes).
- Repeated inheritance. Incompatible superclasses can result in inheriting from one superclass more than once. Solutions:
  1. Graph implementation. Apply all redefinitions from the top down, eliminate identical duplicates, error on non-identical duplicates, resolve by renaming in derived classes.
  2. Linear implementation. Convert graph into a linear list (CPL linearisation), then treat as SI.
  3. Tree implementation. Duplicate common nodes reached by different paths. E.g., C++ with non-virtual base classes.
- Method combination. How should the derived class invoke methods with the same name in different classes? In C++ there is the scope resolution operator (`::`).
- Difficulty of implementation. There have been many attempts, none absolutely right.
- Misuse. MI encourages unrestricted combination of classes even if there is no "is" relationship. E.g., "apple pie" inheriting from both "apple" and "pie", but there is no "is" relationship.

## 9 Uses of Inheritance

The issue is not as simple as SI vs. MI. Behind this binary decision is the more challenging debate of for what to use inheritance.

For example, Booch recommends a *kind* relationship. If B is not a kind of A, then B should not inherit from A. Thus apple pie should not inherit from apple, though inheriting from pie is fine.

### 9.1 Good Uses

Suppose we have a parent class A, and a subclass B. There are many ways we might want to view the relationship between A and B. Often the different views of inheritance lead to the same result, but the conclusions are reached for different reasons.

The following is by Meyer, and gives us a taxonomy of inheritance.

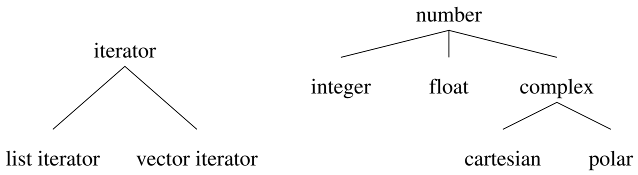


Figure 24: Subtyping

### 9.1.1 Model Inheritance

Classes with an “is a” relationship.

1. Subtype.

   The traditional view of inheritance. Wherever we might use an instance of A we could substitute an instance of B.

   Often A is abstract. For example, `number` and `integer`, or `iterator` and `list_iterator`.

2. View. B describes the same abstraction as A, but from a different perspective. For example: adaptors in C++; interfaces in Java; complex numbers can be Cartesian or polar; a ring can be viewed as a group.

3. Restriction. All the objects in B satisfy some extra condition not specified on A. For example, A is the class of triangles, B is the subclass of equilateral triangles. Cars are a subclass of vehicles as they have 4 wheels, where vehicles don't in general.

4. Extension. B adds behaviour to A which is not appropriate to direct instances of A. This does not contradict restriction:

   - extension refers to attributes and operations
   - restriction refers to instances

   Examples: Cars are a subclass of vehicles as they have radio behaviour; rings extend groups with an extra operator.

### 9.1.2 Variation Inheritance

How classes differ from each other.

5. Functional Variation. B redefines some features of A. B does not redefine any new features above those needed for the redefinition.

   This seems like abuse rather than reuse: A is almost what we need, we just replace some features with new ones.

   Can it be justified? It avoids major system reorganisation, but if possible we should change the hierarchy to define a new abstract class that contains just the parts we want for our class, and then have the old class and our new class inherit from it.

6. Type Variation. Redefining the interface.

7. Uneffecting. B *undefines* some features of A. Being able to do this is very rare. There are two possible justifications:

   (a) under MI we want to merge several concrete classes, and we want to make the superclasses less concrete

   (b) a reusable class does what we want, but contains irrelevant features.

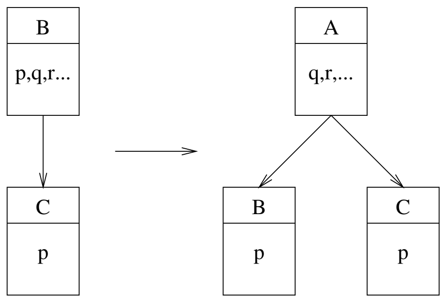


Figure 25: Replacing Functional Variation

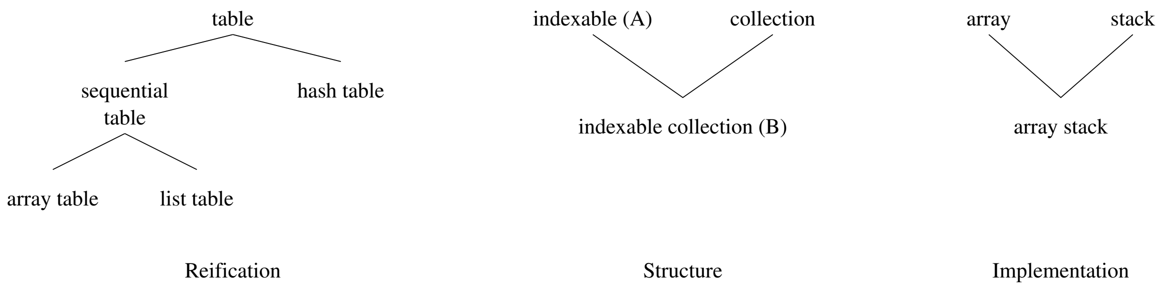


Figure 26: Reification, Structure and Implementation

### 9.1.3 Software Inheritance

Implementational issues.

8. Reification. A is a general form of data structure (abstract). B is a partial or concrete implementation.
9. Structure. A abstract, represents a general property. B represents objects with that property.

   This is one form of *mixin* or *mix-and-match* inheritance. The class A is often an adjective. You might have a colour class that you want to mix in to a car class to get a coloured car class.

   > Aside: "Mixin" as in mixing in flavours in an ice cream. Some people like to restrict multiple inheritance to inheriting from a single base class and multiple mixins. This is because mixins generally don't overlap, and so there are no problems with multiple definitions: a property is only ever inherited from a single class.

10. Implementation. B obtains attributes and operations from A.

    Examples. Building a stack using an array; Java interfaces (array_stack extends array implements stack); STL adaptors (`stack<base_type,representation>`).

11. Facility. A exists only to provide logically related features for subclasses. There are two variants

    (a) constant: where A has constant attributes

    (b) machine: where A has operations (regarded as operations on an abstract machine)

    Examples. Complex as a superclass for Cartesian and polar.

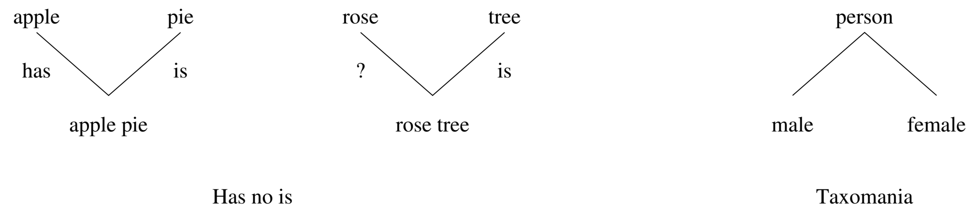


Figure 27: Abuses of Inheritance

### 9.2 Abuses of Inheritance

Meyer has categorised several improper used of inheritance.

- "has" relation with no "is". Apple pie has apple, but is not an apple.

  Aside: "has" relationships are better treated with delegation.

- Taxomania. As in "taxonomy mania". Useless intermediate classes in the hierarchy for trivial values such as booleans. It is generally much better to have a "sex" field in a `person` class rather than create additional `male` and `female` subclasses.
- Convenience inheritance. Simply reusing a class because it is there when there is no true "is" relationship.

  Example. A car class inheriting from a paint class merely because paint has a colour attribute.

## 10 Reflection

### 10.1 What is Reflection?

Consider a Java program running on a machine. The use of Java is purely illustrative: the following is true for any language to a greater or lesser extent.

The compiled program is a sequence of (machine independent) bytes that are interpreted by the Java runtime system. The Java runtime system is merely a program that interprets Java bytestreams, and is generally machine specific. The runtime system is compiled machine code, which on some processors is interpreted in microcode on the CPU. The microcode interpreter runs on the hardware, the silicon itself.

Each cloud represents a form of program.

- looking down, a program sees the application it is running
- lookup up, a program sees the processor that is running it.

Reflection is the ability of a program being able to reason about and influence its processor. The program must be able to

- extract information from the processor in a form it can manipulate. This is called *reification*. It must also be able to
- insert information into the processor in a form it can use. This is called *reflection*.

Why should we want to do this? It facilitates certain kinds of application:

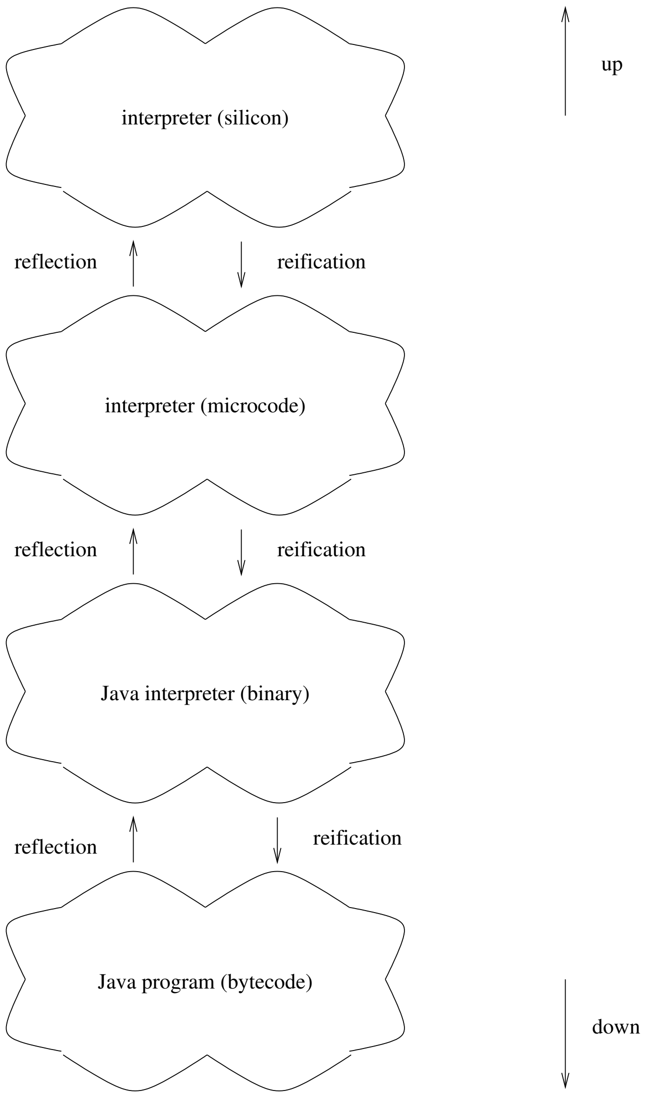


Figure 28: Reflection and Reification

- generic tools that need access to the slots of an object, e.g., printers, object inspectors
- debuggers that need run time access to all slots
- interpreters that add another program to the tower
- serialisers: these convert objects to a format that can be transmitted between programs or over networks

The level of reflective access is a measure of the flexibility of a language and how much the programmer can

> program the programming language

## 10.2 Javascript

Reification operations include

- `toString` method. Amongst others we get `int` → `string`, `float` → `string`, `function` → `string`, though the latter loses a lot. The user is expected to define a `toString` for their own objects.
- slot access:

  `for (i in obj) { ... }`

  `obj.slot vs obj['slot']`

- `in` operator: `"colour" in c` is true if `c` has a slot named `colour`
- `instanceof` operator: if `s = new String("fred")` then `s instanceof String` returns true.

There is one reflection operation:

- `eval`: this takes a string and evaluates it as a Javascript expression. Thus `eval("1+2")` returns 3.

See Javascript handout.

## 10.3 Java

There is a limited reflection API. Reification operations include

- getting information about members and constructors
- getting and setting attribute values
- calling methods
- creating new instances

Java allows construction of new instances of the new classes `Field`, `Method`, `Constructor`.

See Java handout.

Java has no reflection operations.

### 10.4 C++

C++ offers us very little. It is limited to dynamic access to type information.

#### 10.4.1 `Typeid`

`typeid(x)` returns an object of type `type_info`. Defined operations on this object are

- `==`, `!=`, i.e., equality `if (typeid(a) == typeid(b)) ...`
- `name` to get a string which names the type of the object: `typeid(x).name()`
- `before` to get the ordering of subclasses: `typeid(a).before(typeid(b))` is true if the class of `a` is a subclass of the class of `b`.

`typeid` only works on classes with a virtual method.

> Aside: `typeid` works like this: a `type_info` structure is created (one per class), that contains the `name` and so on. An extra hidden slot in each instance points to the structure.

#### 10.4.2 `dynamic_cast`

We can assign a `car` to a variable of type `vehicle`, as this always makes sense. On the other hand, it is dangerous to assign a `vehicle` to a `car` variable: maybe it is is really a car, maybe it is not. Thus C++ allows `cars` to be assigned to `vehicles`, but not vice-versa. We can always use a C-style cast to force the compiler, but then we get what we deserve.

The `dynamic_cast` is a means to get this right for us.

```
vehicle *v;
car *c;
...
c = dynamic_cast<car*>v;
```

If `v` is actually an instance of `car*`, this will assign it to `c`. If `v` is not an instance of `car*`, this will assign `NULL` to `c`.

Thus this is a type-safe *downcast* operation. The opposite, *upcasting*, is always safe.

> Aside: `dynamic_cast` works like this: the `type_info` structures for both source and destination types are fetched. A library routine then determines whether the upcast is safe by comparing the superclass hierarchies.

## 10.5 Lisp

Lisp offers us the state-of-the-art reflection. Common Lisp has CLOS (Common Lisp Object System), and EuLisp has Telos (The EuLisp Object System). Both are fully reflective, providing a *meta-object protocol* (MOP). The idea of a MOP came from ObjVlisp.

A meta-oject protocol is a means to manipulate meta-objects, i.e., objects that describe objects, namely classes.

Early versions of Lisp provided only the unstructured reflection and reification found in Javascript. CLOS is notable for being the first attempt to design a fully reflective system without compromising performance.

> Aside: *Uncontrolled reflection* means that anything can be changed, so compilation is pretty much impossible.

CLOS defines precisely the very many but carefully chosen places where the user can reify and reflect: there are too many to list in full here, but we get methods to inspect classes:

- `class-subclasses`
- `class-superclasses`
- `class-instance-length`
- `class-name`
- `compute-slot-reader`
- `compute-slot-writer`
- `class-precedence-list`
- `class-slot-descriptions`
- `subclassp`

methods to inspect generic functions

- `find-method`
- `generic-function-domain`
- `generic-function-methods`

methods to inspect methods

- `method-domain`
- `method-generic-function`
- `method-function`

methods to inspect slot in objects

- `slot-description-name`

- `slot-description-slot-reader`
- `slot-description-slot-writer`

and others

- `class-of`
- and so on.

The above is just a sample.

> Aside: The really amusing part is the implementation of MOPs. What is the best language to implement a dynamic behaviour like a MOP? It turns out that the best way to implement CLOS in Common Lisp is to use CLOS, and the best way to implement Telos in EuLisp (or, indeed Common Lisp) is to use Telos. There are some bootstrapping problems to overcome, but once these are addressed, everything goes very well, as Telos provides exactly the functionality needed to implement a language like Telos. Ditto CLOS for CLOS.

## 10.6 Summary

Reflection blurs the distinction between

- language designer
- programmer

and allows the latter to adjust the language better to suit their needs. It also permits the construction of generic tools and programming environments for object oriented languages.