System Modelling

* Based on notes by Brad Lushman, used with ... I'll ask later.

We want to visualize the structure of the system we are trying to build.

- Abstractions and relationships among them.
- Aid in design and implementation - the visualization is easier to understand the system than understanding all of the code.

UML - Unified Modelling Language
https://student.cs.uwaterloo.ca/ cs246/S23/resources.shtml

A class in UML is drawn as a box with 3 sections:

- 1. Class Name
- 2. (Optional) Fields: `< <Access Specifier> Name : Type >`
- 3. (Optional) Methods: `< <Access Specifier> Name() : Type >`

Access Specifiers: '-' represents private, '+' represents public

Relationship: Composition of classes

Embedding an object B inside another object A where B's only purpose is to be used in A. A *"owns a"* B.

If A *owns a* B, then typically

- B has no identity outside A - no independent existence
- if A is destroyed, then B is destroyed
- if A is copied, B is copied (deep copy)

Notation: A arrow with a solid diamond tail pointing at B.
Arrow is annotated with field names and multiplicities (1, 2, 0..*, etc).

Relationship: Aggregation

Embedding an object B inside another object A but B exist on its own. A *"has a"* B.

If A *has a"* B, then typically

- B exists apart from it's association with A
- if A is destroyed, B lives on
- if A is copied, B is not (shallow copy) and copies of A will share the same B

Notation: A arrow with an unfilled diamond tail pointing at B.

Case Study

Does a pointer field always mean non-ownership?

No! Consider `Lists` and `Nodes`.

A `Node` owns the `Nodes` that follow it - implementation of Big 5 is a good indication of ownership. Then a `List` owns the first `Node`.
These ownerships are implemented by pointers.

Case Study

Does a pointer field always mean non-ownership?

No! Consider `Lists` and `Nodes`.

A `Node` owns the `Nodes` that follow it - implementation of Big 5 is a good indication of ownership. Then a `List` owns the first `Node`.
These ownerships are implemented by pointers.

Alternatively, you could view the `List` as owning all the `Nodes` within it. Then, the `List` is likely taking responsibility for copying and destroying all of the `Nodes`, rather than `Node`.
$\Rightarrow$ `List` may use an iterative (loop-based) implementation to manage pointers rather than a recursive one where `Nodes` manage other `Nodes`.

Relationship: Inheritance

B "is a" A

- *A* is called a Base class or Superclass
- *B* is called a Derived class or Subclass

Derived classes *inherit* fields and methods from the base class.
Any method that can be called on the Base class can be called on the derived class.

Rules: public inheritance ...