Object-oriented programming

Lecture #3: Introduction to OOP

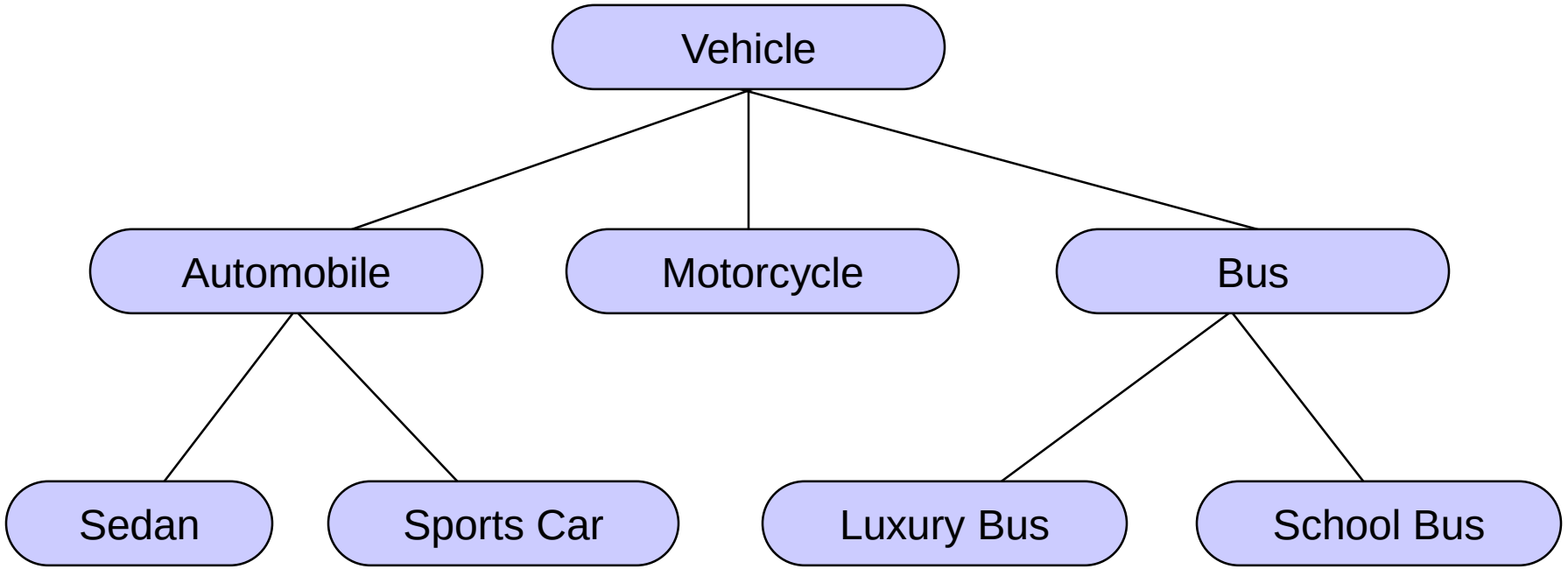
Solving a Programming Problem

- Analysis
- Design
- Coding
- Management

- Programming paradigms
- Programming languages

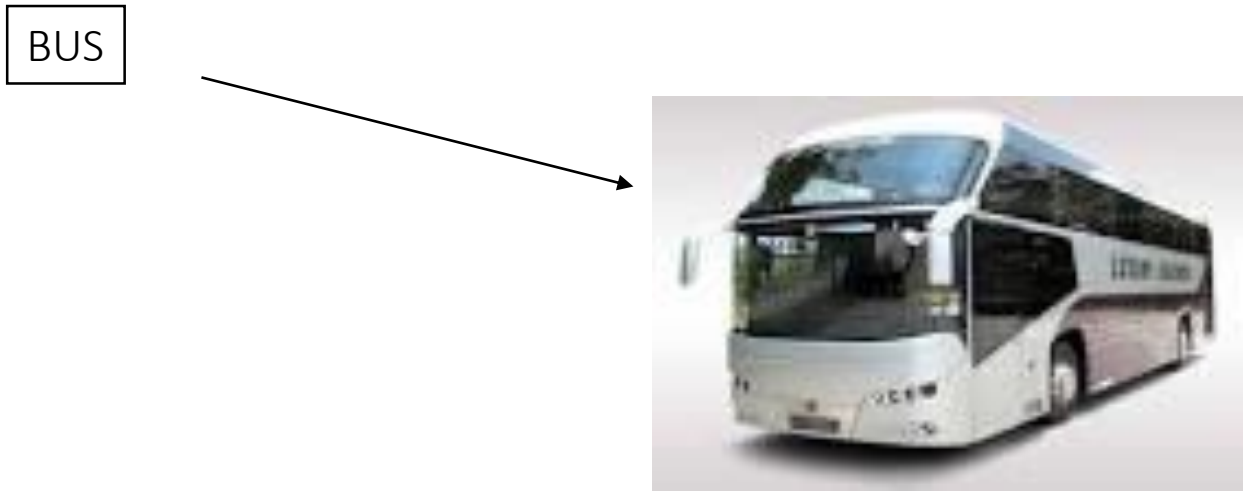
Thinking Methodology

- Induction
  - From specialization to generalization
  - E.g., to create the (abstract) concept “Bus” from different bus



Thinking Methodology

- Deduction (infer)
  - From generalization to specialization
  - E.g., from the concept “Bus” we have learned, we deduce that a vehicle is or is not a bus.



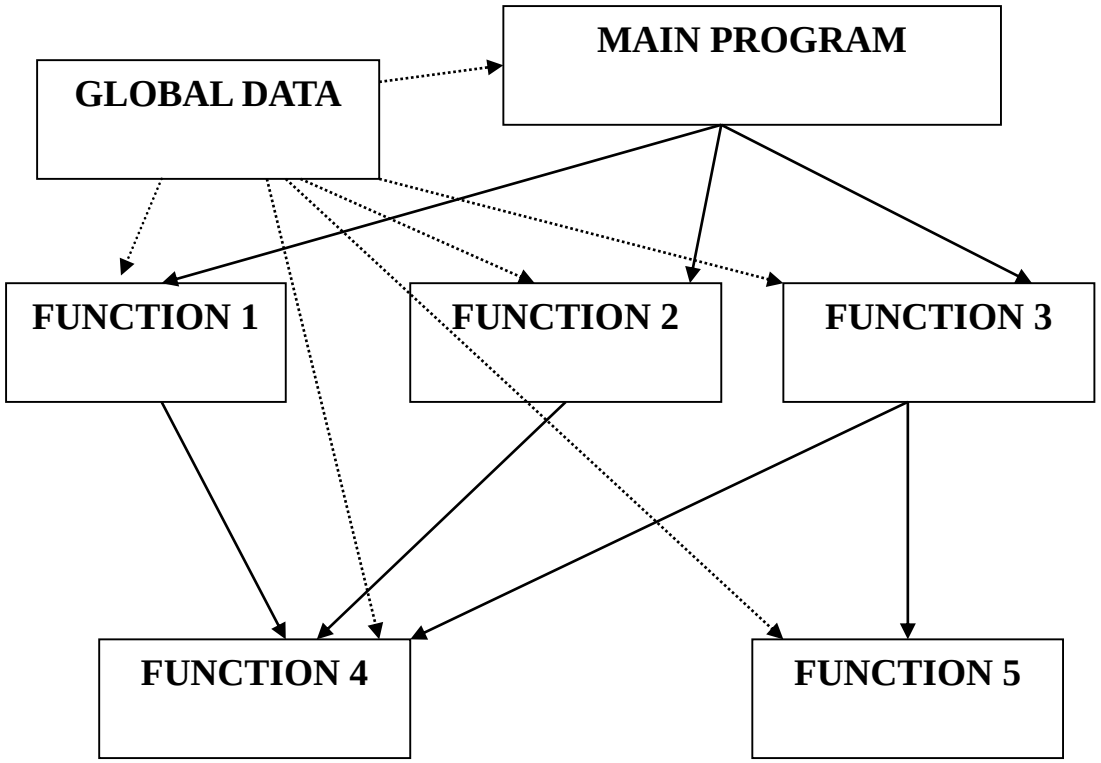
Concepts and Relationships

- A rectangle uses lines
- A circle is an ellipse
- A wheel is part of automobile
- A set creates its elements

Approaches in Software Development

- Top Down approach
  - A Single module will be split into several smaller modules
  - General to Specific
- Bottom Up approach
  - Lot of small modules will be grouped to form a single large module
  - Specific to General
- If the requirements are clear at the first instance we can go for Top down approach
- In circumstances where the requirements may keep on adding, we go for Bottom up approach

Structured programming



- Using function
- Function & program is divided into modules
- Every module has its own data and function which can be called by other modules.

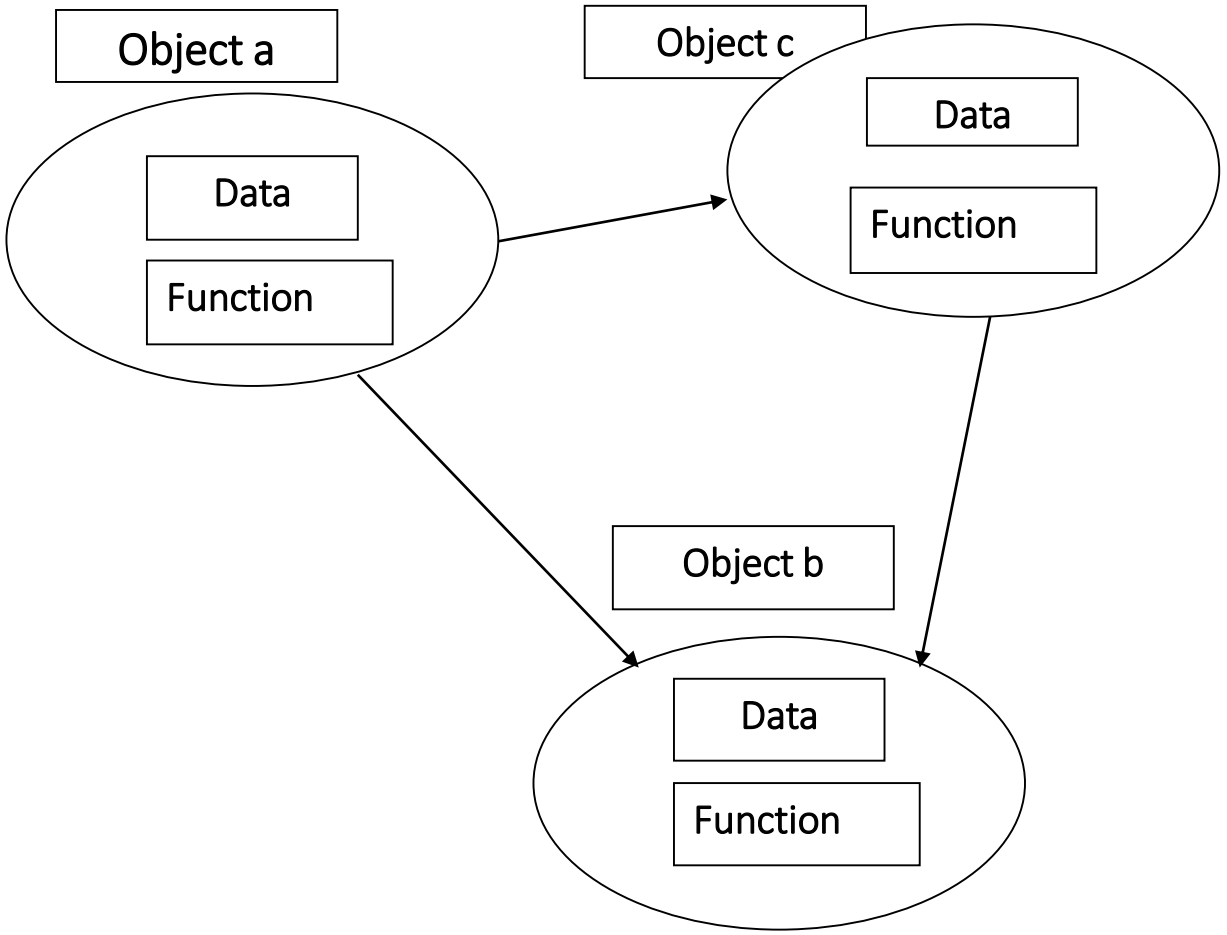
OOP Significant

- In real world scenario, for developing a software, we need requirements.
- But practically speaking, requirements keep on changing and request to add new features will keep on accumulating.
- So its better if we follow the Object Oriented Programming Strategy (bottom up approach)

Object-Orientation

- A thinking methodology
  - Everything is an object
  - Any system is composed of objects (a system is itself an object)
  - The evolution and development of a system is caused by the interactions of the objects inside/outside a system

Object-Orientation



- Objects have both data and methods
- Objects of the same class have the same data elements and methods
- Objects send and receive *messages* to invoke actions

Key idea in object-oriented:

- *The real world can be accurately described as a collection of objects that interact.*

Object-Orientation

- Everything is an object
  - A student, a professor
  - A desk, a chair, a classroom, a building
  - A university, a city, a country
  - The world, the universe
  - A subject such as CS, IS, Math, History, …

Object-Orientation

- Systems are **composed of** Objects
  - An education system
  - An economic system
  - An information system
  - A computer system

Object-Orientation

- The development of a system is caused by interactions
- E.g., VGU is defined by the interactions among:
  - students
  - professors
  - staff
  - board governance
  - state governance
  - …

Inside VGU

Outside VGU

Object-Orientation

- Object-Orientation is a design methodology (OOD)
  - Objects are the building blocks of a program
  - Objects represent real-world abstractions within an application

Design Methodology

- Object-orientation supports
  - Induction: objects → a class
    - *There are tools doing this automatically*
  - Deduction: a class → objects
    - *Usually done by programmers*

Design Methodology

- Object-orientation supports
  - Top-down: from a super-class to sub-classes
  - Bottom-up: from sub-classes to a super-class

Why OOP?

- An OOP language should support
  - Easy Representation of
    - *Real-world objects*
    - *Their States and Abilities*
  - Interaction with objects of same type
  - Relations with objects of other type
  - Polymorphism and Overloading
- Convenient type definitions

Why OOP?

- Save development time (and cost) by reusing code
  - once an object class is created it can be used in other applications
- Easier debugging
  - classes can be tested independently
  - reused objects have already been tested

Design principles of OOP

- Four main design principles of Object-Oriented Programming(OOP):
  - *Abstraction*
  - *Encapsulation*
  - *Polymorphism*
  - *Inheritance*

Abstraction

- Focus only on the important facts about the problem at hand to design, produce, and describe so that it can be easily used without knowing the details of how it works.

Analogy:

- When you drive a car, you don’t have to know how the gasoline and air are mixed and ignited. Instead you only have to know how to use the controls.

Abstract Data Types (ADT)

- In ADTs: to define the interface to data abstraction without specifying implementation details
- ADT includes the following properties
  - It exports a type
  - It exports of set of operations
  - Axioms and preconditions define the application domain of the type

ADT in C++

- Showing only the essential features and hiding the unnecessary features
- The access modifiers in C++ or any OOP language, provides abstraction
- Functions also provide abstraction
- If a variable is declared as private, then other classes cannot access it
- The function name that is used in function call hides the implementation details from user. It shows only the outline of functionality that the function provides.

Encapsulation

- The process of bringing together the data and method of an object is called as encapsulation
- The data and method are given in the class definition
- Classes provides us with this feature - Encapsulation

Encapsulation

- Allows for modularity
- Controls access to data
- Separates implementation from interface
- Extends the built-in types

Example

- Complex numbers:
  - Real part
  - Imaginary part

  Both parts are represented by real numbers

  - Operations: addition, substraction, multiplication or division to name a few
- To represent a complex number it is necessary to define the data structure to be used by its ADT.

Example (Complex numbers)

- One can think of at least two possibilities to do this:
  - Both parts are stored in a two-valued array where the first value indicates the real part and the second value the imaginary part of the complex number. If *x* denotes the real part and *y* the imaginary part, you could think of accessing them via array subscription: $x=c[0]$ and $y=c[1]$.
  - Both parts are stored in a two-valued record. If the element name of the real part is *r* and that of the imaginary part is *i*, *x* and *y* can be obtained with: $x=c.r$ and $y=c.i$.

⇒ In the first version, *x* equals $c[0]$. In the second version, *x* equals $c.r$. In both cases *x* equals "something"". It is this "something" which differs from the actual data structure used. But in both cases the performed operation "equal" has the same meaning to declare *x* to be equal to the real part of the complex number *c*: both cases archieve the same semantics.

⇒ the ADT definition says that for each access to the data structure there should be an operation defined => contradicted!

Example (Complex numbers)

- Once you have created an ADT for complex numbers, say *Complex*, you can use it in the same way like well-known data types such as integers.
  - *Complex* a;

Example (Complex numbers)

- If you think of more complex operations the impact of decoupling data structures from operations becomes even more clear.
  - For example the addition of two complex numbers requires you to perform an addition for each part. Consequently, you must access the value of each part which is different for each version.
  - By providing an operation “add” you can *encapsulate* these details from its actual use. In an application context you simply “*add two complex numbers*” regardless of how this functionality is actually archieved.

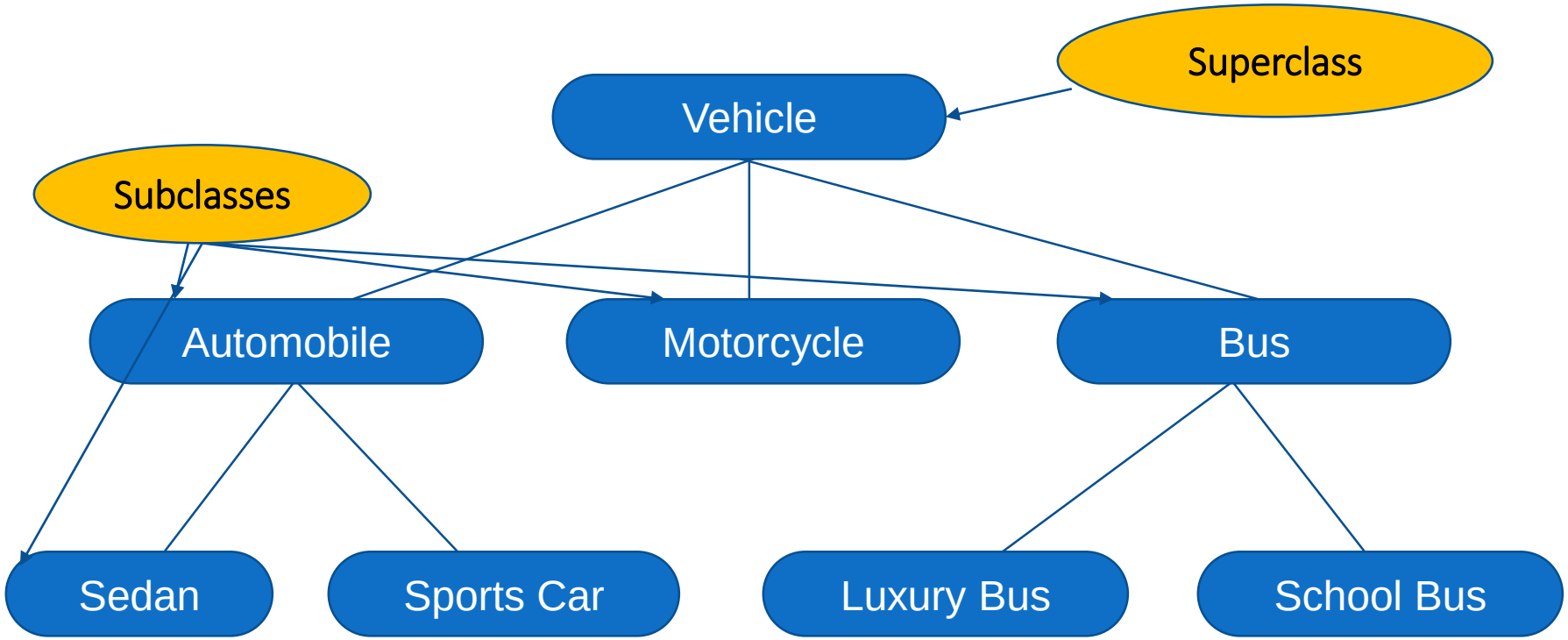
Inheritance

- Allows code reusability, by which a new abstract data type can inherit the data and functionality of some existing type, and is also allowed to modify some of those entities and add new entities.

Inheritance Terminology

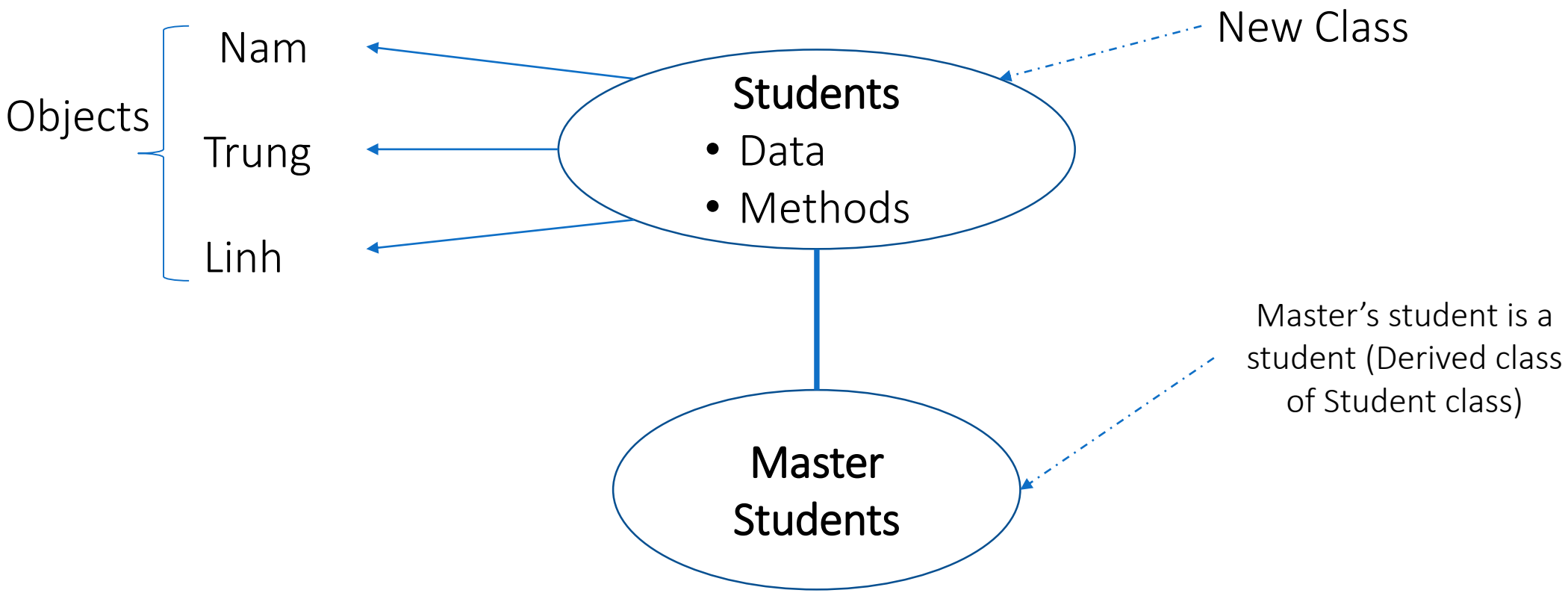
- **Class:** the abstract data type in Object-Oriented languages.
- **Object:** A class instance.
- **Derived class (subclass):** A class that is defined through inheritance from another class.
- **Parent class (superclass):** A class from which a new class is derived.
- **Methods:** the subprograms that define the operations on objects of a class.
- **Messages:** the calls to methods. Messages have two parts--a method name and the destination object.
- **Message Protocol (Message Interface):** the entire collection of an object's methods.

Inheritance



What properties does each vehicle inherit from the types of vehicles above it in the diagram?

Inheritance



Polymorphism

- The ability of objects to respond differently to the same message or function/method call
  - *Poly – Many*
  - *Morph – Form*
  - *Polymorphism => co exist in more than one form*
- C++ supports function overloading and operator overloading to implement polymorphism