# ECE326

# PROGRAMMING LANGUAGES

**Lecture 7 : Inheritance and Runtime Polymorphism**

Kuei (Jack) Sun

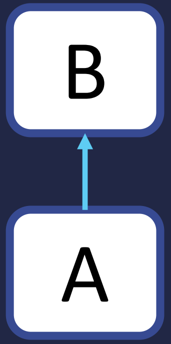
ECE

University of Toronto

Fall 2020

# Inheritance

- Creates new class (subclass) based on existing one(s)
  - Acquires all attributes and behaviours of parent (base class)
  - Retains all existing implementation
    - Enables code reuse
  - Can extend to support new behaviours
    - Add more functionality
  - Can replace (override) existing implementation
- Used interchangeably
  A. Subclass – child class – derived class
  B. Super class – parent class – base class



# Inheritance in Python

```
class Animal:
    def __init__(self, age, height):
        self.age, self.height = age, height
    def move(self, location):
        print(“It moved to %s”%location)

class Dog(Animal):
    def __init__(self, age, weight, name=“Marley”):
        self.age, self.height = age, height
        self.name = name
    def move(self, location):
        print(“%s moved to %s”%(self.name, location))

>> dog = Dog(5, 35.2)
>> dog.move(“the park”)
Marley moved to the park
```

# super

- Refers to the super class(es) of this class
- Do not have to specify which one
  - Very important for multiple inheritance (defer to later lecture)

```
class Dog(Animal):
    def __init__(self, age, weight, name=“Marley”):
        Animal.__init__(self, age, weight)
        self.name = name
```

```
class Dog(Animal):
    def __init__(self, age, weight, name=“Marley”):
        super().__init__(age, weight)
        self.name = name
```

## Runtime Polymorphism

- Choosing behaviour through single interface at runtime
  - Dynamic Dispatch
    - Decides which implementation of a virtual function to call
  - Late Binding
    - Performs name binding at runtime
    - E.g. Python’s attribute name resolution
- There are no non-virtual methods in Python
  - Every attribute can be overridden by child class
    - Sometimes causes unintentional override

# Dynamic Dispatch

- A *polymorphic* operation with different implementations
- Determines which to call at *runtime* based on *context*
  - Context can include caller’s type and input types
- Static Dispatch
  - Ad-hoc polymorphism
  - Can be done at compile-time
  - Knows which function to call based on their signatures
- Can mix dynamic and static dispatch
  - E.g. virtual operator overloading

## Dynamic Dispatch

- C++ virtual functions
- Context is based solely on type of instance
  - *Not* the reference type of the variable

```
struct A {
     virtual void foo() {
           cout << “A::foo\n”;
     }
};

struct B : public A {
     virtual void foo() override {
           cout << “B::foo\n”;
     }
};
```

```
B b = B();

// reference type is A
// instance type is B
A * ap = &b;

// prints B::foo
ap->foo();
```

## Late Binding

- Associates *name* with an *attribute* at runtime
  - Also known as dynamic binding
  - Type *unknown* until use (e.g., evaluation)
- Early Binding
  - Also known as static or compile-time binding

## Binding vs. Dispatch

- Late binding is concerned with the *object*
  - Calling method by *name*
  - Name resolved to attribute by *object type*
  - Object behaviour can change after instantiation
- Dynamic dispatch is concerned with the *operation*
  - Calling method by *context*
    - Context determines which implementation to call
  - Object behaviour remains the same after instantiation