Inheritance

Table of contents

Outline 1
Inheritance 1
Terminology 2
Reusing code via inheritance 4
Polymorphism 6
Exercise 7
Summary 8

See csse1001[1] for course logistics. This is Lecture 9B. See python-inheritance[2] for the full reference on subclassing, `super()`, and polymorphism.

Outline

- Inheritance and "is-a" relationships
- Terminology: subclass/superclass, child/parent, extends
- Inheriting, introducing, and overriding methods
- `super()` and extending methods
- Polymorphism

Inheritance

Inheritance is another way of reusing code in classes. By inheriting from a class, we can create specialized (sub)classes while reusing attributes/methods from the original class. Key idea: building a **hierarchy of classes**.

This is an **"is-a"** relationship: a `DeliveryRobot` *is a* `Robot`. `DeliveryRobot`, `CleaningRobot`, and `HumanoidRobot` all **specialize** the base class (`Robot`):

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-inheritance.pdf

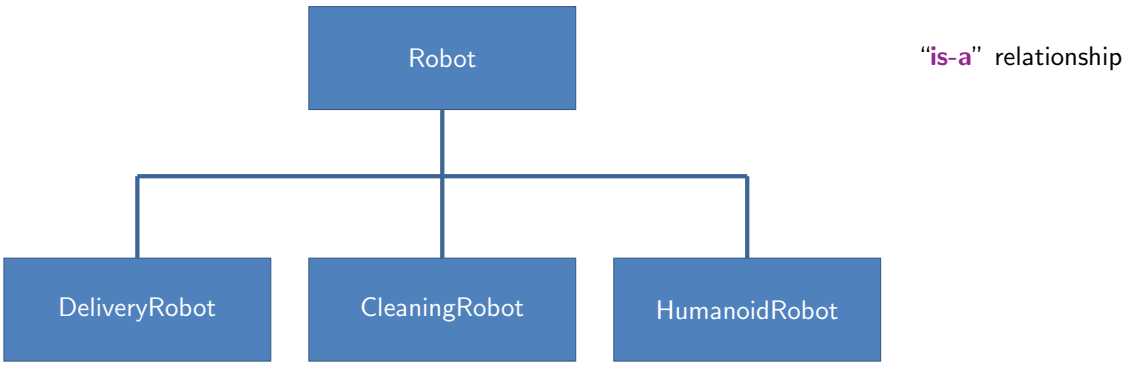

Figure 1: The `Robot` hierarchy. Parent above, children in a row beneath, joined by a single stem down to a horizontal bus with a drop to each child. The connector carries no arrowhead; the next figure draws inheritance with arrows instead.

- They **inherit** the attributes and methods of the base class.
- They can **override** the attributes/methods of the base class — for example, `move()` might have a different implementation in each.
- They can have **new (unique)** attributes/methods, or new implementations.

More is-a hierarchies

Three more is-a hierarchies:

- `Employee` → `Programmer`, `Manager`, `Admin`
- `Animal` → `Cat`, `Dog`, `Fox`; and below `Dog`, `ServiceDog`
- `Person` → `Professor`, `Student`; and below `Student`, `UndergradStudent` and `GradStudent`

See csse1001-diagram-notation[3] for the two ways of drawing the inheritance connector, and 2025-10-07-uml[4] for the formal notation that supersedes both in week 10.

Terminology

Given `class Student(Person):` — all of the following mean the same thing, and can be used interchangeably:

- The `Student` class **inherits** from the `Person` class.
- The `Student` class is a **subclass** of the `Person` class.

[3] courses/csse1001/csse1001-diagram-notation.pdf
[4] courses/csse1001/2025-10-07-uml.pdf

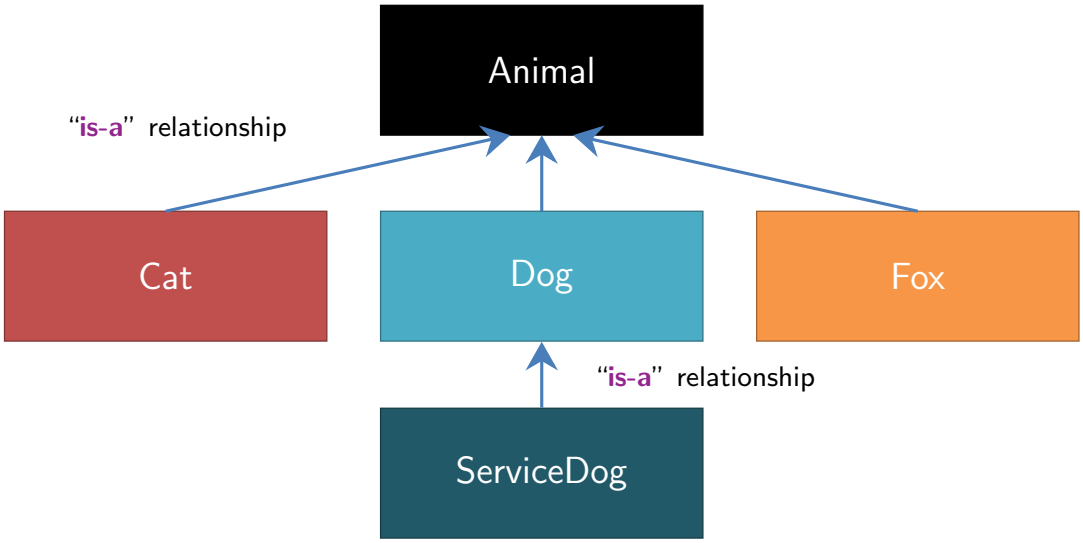

Figure 2: The `Animal` hierarchy, drawn in a different style from Figure 1. Here the root is black, each child gets its own fill, and the connectors are thin straight lines — two diagonal, one vertical — each ending in its own arrowhead on the parent's bottom edge. `ServiceDog` shows that a child can itself be a parent: it gets its own arrow into `Dog`. The child fill colours carry no meaning — `Dog` stays teal even though it is itself a parent.

- The `Student` class is a **child class** of the `Person` class.
- The `Person` class is a **superclass** of the `Student` class.
- The `Person` class is a **parent class** of the `Student` class.
- (less often) The `Student` class **extends** the `Person` class.

Reusing code via inheritance

Start with a base `Robot`, and a `CleaningRobot` written without inheritance:

```
class Robot:
    def __init__(self, name: str, batt_level=100):
        self.name = name
        self.batt_level = batt_level

    def charge(self):
        self.batt_level = 100
        print(f"{self.name} is fully charged!")


# CleaningRobot without inheritance -- repeated code
class CleaningRobot:
    def __init__(self, name: str, batt_level: int):
        self.name = name
        self.batt_level = batt_level

    def charge(self):
        self.batt_level = 100
        print(f"{self.name} is fully charged!")

    def clean(self):
        print(f"{self.name} is cleaning the floor.")
```

The second listing repeats two members of `Robot`: `charge` and `__init__`. Only `clean()` is genuinely new. Instead, **inherit from `Robot`**:

```
# Inherit specialized robots from the base class
class CleaningRobot(Robot):
    def clean(self):
        print(f"{self.name} is cleaning the floor.")
```

That is the entire class. The syntax `class Child(Parent):` is all it takes — see python-inheritance[5] for the annotated form of that line.

`CleaningRobot` now inherits **all attributes**, `__init__`, and `charge` from `Robot`; constructing one runs `Robot.__init__`, and `cleaner.name` and `cleaner.charge()` both work without `CleaningRobot` mentioning them.

The four things a subclass can do

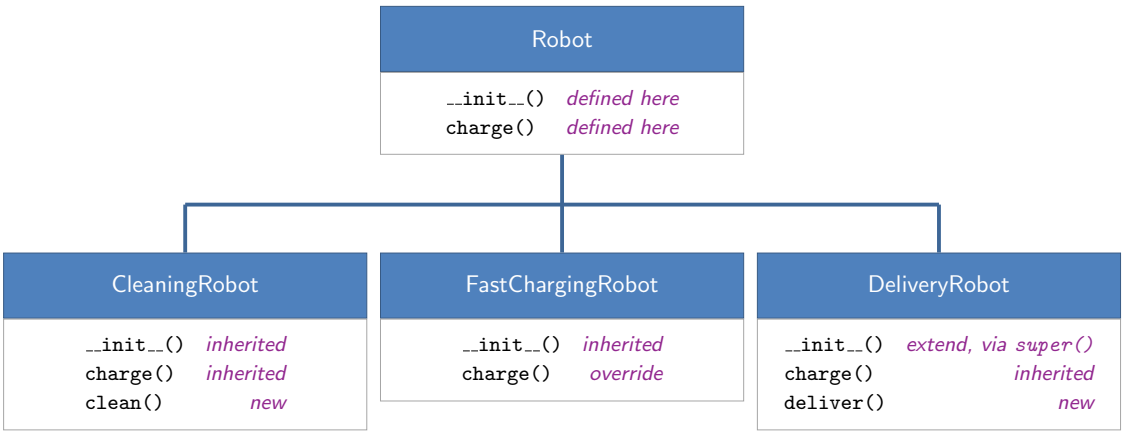

Figure 3: What each subclass does with each member of `Robot`: inherit it, override it, extend it, or add a new one. Each box lists the class's members. The purple italic annotation beside each subclass member says which of the four applies; `Robot`'s own members are marked as defined there. The three-compartment UML class box that does this properly comes in week 10.

Introducing new methods

A child class can define methods that don't exist on the base class at all. `clean()` only exists on `CleaningRobot` objects — calling it on a plain `Robot` is an error.

Overriding methods

We **override** a method by declaring one of the same name in the child class, exactly as we would normally declare a method:

[5] courses/csse1001/python-inheritance.pdf

```
class FastChargingRobot(Robot):
    def charge(self):   # overriding the charge method from the base class
        self.batt_level = 100
        print(f"{self.name} is fully charged in record time!")
```

Calling `charge()` on a `FastChargingRobot` object calls *this* method, not `Robot`'s. Nothing else changes: `FastChargingRobot` still inherits `__init__` and `name`.

Extending methods with `super()`

`super()` specifies that we want to use the method from the **superclass**, rather than the child class. That is what makes it possible to *extend* inherited behaviour instead of replacing it — most often in `__init__`, where the parent already knows how to set up the attributes it defined:

```
class DeliveryRobot(Robot):
    def __init__(self, name: str, batt_level: int, load_capacity: int):
        # super().__init__(...) calls the __init__ method from the parent (base) class
        super().__init__(name, batt_level)
        self.load_capacity = load_capacity

    def deliver(self):
        print(f"{self.name} is delivering a package up to {self.load_capacity}kg.")
```

`DeliveryRobot.__init__` handles only what is new (`load_capacity`) and hands `name` and `batt_level` back to `Robot.__init__`. This signature drops the `batt_level=100` default that `Robot.__init__` has — a `DeliveryRobot` must be given a battery level.

Polymorphism

The word **polymorphism** means to take on many forms. In computer science, an **interface** that works over different underlying data-types is called **polymorphic**.

The `len` function is polymorphic:

```
>>> len({'a': 1, 'b': 2, 'c': 3})
3

>>> len("Drop Bear")
9
```

```
>>> len([1, 2, 3, 4])
4
```

Classes in one hierarchy are polymorphic in exactly the same way: the same call means "do the right thing for whatever this object actually is".

```
# Polymorphism
robots = [
    CleaningRobot("Roomba", 40),
    DeliveryRobot("DHL-Bot", 20, 15),
    FastChargingRobot("FlashBot", 5)
]

for r in robots:
    r.charge()   # behavior depends on class of object
```

The list holds three different classes, and the loop body does not test which is which. What each iteration runs is decided by the object, not by the loop. Following each call through the class definitions:

- `CleaningRobot` → no `charge` of its own → `Robot.charge` → *"Roomba is fully charged!"*
- `DeliveryRobot` → no `charge` of its own → `Robot.charge` → *"DHL-Bot is fully charged!"*
- `FastChargingRobot` → has its own `charge` → stops there → *"FlashBot is fully charged in record time!"*

Adding a fourth robot class later does not require touching this loop — that is what the shared interface buys.

Exercise

Implement the `SurveyDrone` and `DeliveryDrone` classes, inheriting from the `DroneFlight` class of 2025-09-23-composition[6]:

```
>>> class DroneFlight:
...     pass # Implemented before
>>> class SurveyDrone(DroneFlight):
...     pass # Implement SurveyDrone
>>> class DeliveryDrone(DroneFlight):
...     pass # Implement DroneFlight
```

[6] courses/csse1001/2025-09-23-composition.pdf

Both subclasses are left as stubs to implement. The last comment should read `# Implement DeliveryDrone`.

The exercise uses both techniques: `DroneFlight` was built by *composition* in the previous lecture, and these two subclasses extend it by *inheritance*. The two techniques are not alternatives at the level of a program — a `SurveyDrone` **is a** `DroneFlight`, which **has a** `Camera`.

Summary

We can create classes that **inherit** from other classes. By doing so, the subclass automatically receives all attributes and methods implemented in the superclass. The subclass can have additional methods and attributes, while **overriding** inherited methods and **calling** methods from its superclass.

Use inheritance only when there's a clear **"is-a"** relationship.

Next: 2025-10-07-uml[7] (Week 10), which gives this hierarchy a formal notation, and 2025-10-07-advanced-inheritance[8], which takes `super()` further.

[7] courses/csse1001/2025-10-07-uml.pdf
[8] courses/csse1001/2025-10-07-advanced-inheritance.pdf