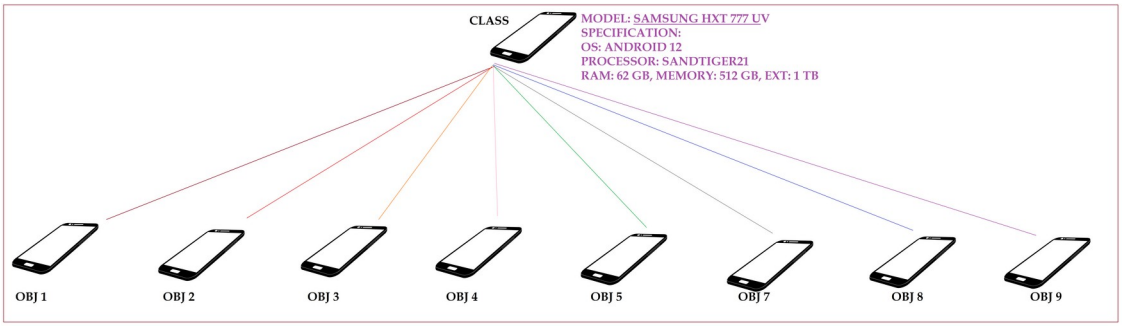
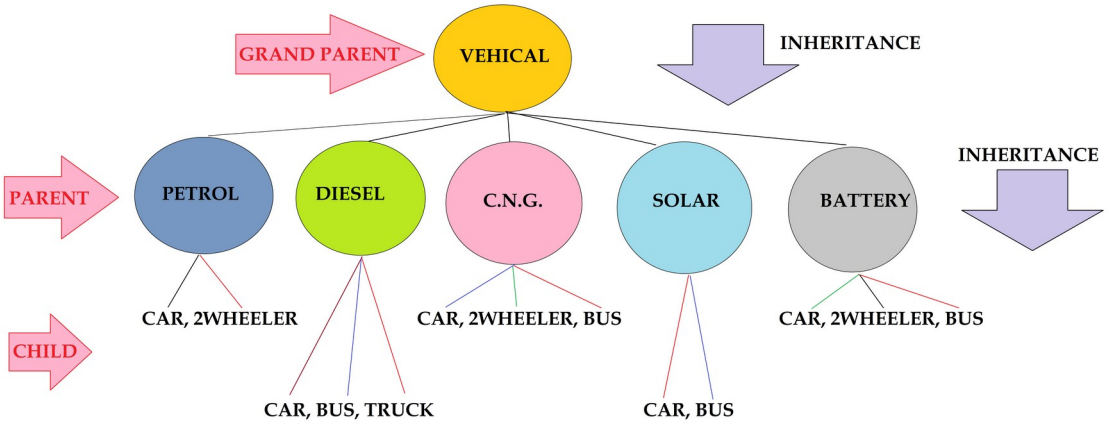
TheDataLytics

python™

ANUURAG EDLABADKAR





Literal Meaning of Inheritance: Inheritance is the practice of passing on private property, titles, debts, entitlements, privileges, rights, and obligations upon the death of an individual. The rules of inheritance differ among societies and have changed over time. The passing on of private property and/or debts can be done by a notary.

Inheritance

In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones such as super class or base class and then forming them into a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance, a "child object", acquires all the properties and behaviors of the "parent object" , with the exception of: constructors, destructor, overloaded operators and friend functions of the base class. Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a directed acyclic graph.

Inheritance was invented in 1969 for Simula and is now used throughout many object-oriented programming languages such as Java, C++, PHP and Python.

An inherited class is called a subclass of its parent class or super class. The term "inheritance" is loosely used for both class-based and prototype-based programming, but in narrow use the term is reserved for class-based programming (one class inherits from another), with the corresponding technique in prototype-based programming being instead called delegation (one object delegates to another).

Inheritance should not be confused with subtyping. In some languages inheritance and subtyping agree, whereas in others they differ; in general, subtyping establishes an is-a relationship, whereas inheritance only reuses implementation and establishes a syntactic relationship, not necessarily a semantic relationship (inheritance does not ensure behavioral subtyping). To distinguish these concepts, subtyping is sometimes referred to as interface inheritance (without acknowledging that the specialization of type variables also induces a subtyping relation), whereas inheritance as defined here is known as implementation inheritance or code inheritance. Still, inheritance is a commonly used mechanism for establishing subtype relationships.

Inheritance is contrasted with object composition, where one object contains another object (or objects of one class contain objects of another class); see composition over inheritance. Composition implements a has-a relationship, in contrast to the is-a relationship of subtyping.

```
class Citizen:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Citizen class to create an object, and then execute the
```

printname method:

```
x = Citizen("Ram", "Shyam")
x.printname()
```

```
Ram Shyam
```

```
class Citizen:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Pupil(Citizen):
  pass

x = Pupil("Indresh", "Indu")
x.printname()
```

```
Indresh Indu
```

```
class Citizen:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Pupil(Citizen):
  def __init__(self, fname, lname):
    Citizen.__init__(self, fname, lname)

x = Pupil("Indresh", "Indu")
x.printname()
```

```
Indresh Indu
```

```
class Citizen:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Pupil(Citizen):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

```
x = Pupil("Indresh", "Indu")
x.printname()
```

```
Indresh Indu
```

```
class Citizen:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Pupil(Citizen):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019

x = Pupil("Indresh", "Indu")
print(x.graduationyear, x.firstname, x.lastname)
```

```
2019 Indresh Indu
```

```
class Citizen:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Pupil(Citizen):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)

x = Pupil("Indresh", "Indu", 2019)
x.welcome()
```

```
Welcome Indresh Indu to the class of 2019
```

```
class Citizen:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
```

```
  def printname(self):
    print(self.firstname, self.lastname)

class Pupil(Citizen):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Pupil("Indresh", "Indu", 2019)
print(x.graduationyear)
```

```
2019
```