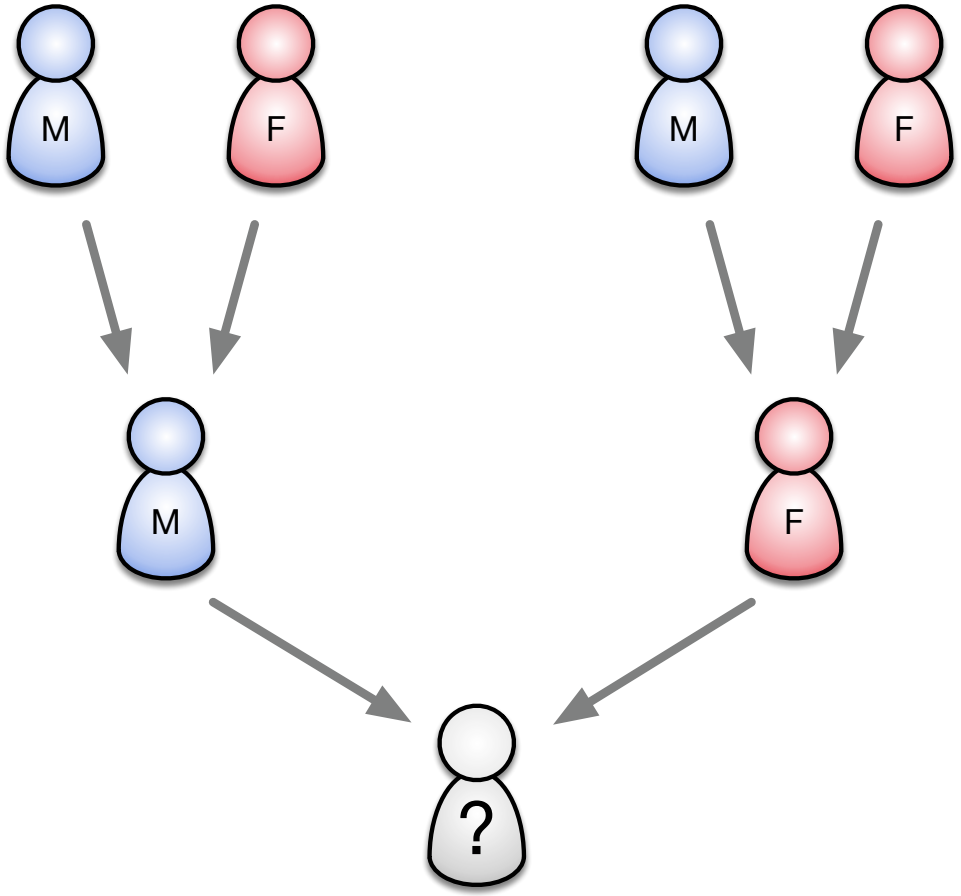
Inheritance

Prof. Dr. Harald Gall

University of Zurich, Department of Informatics

Family Tree



Specific traits are passed on in a family tree.

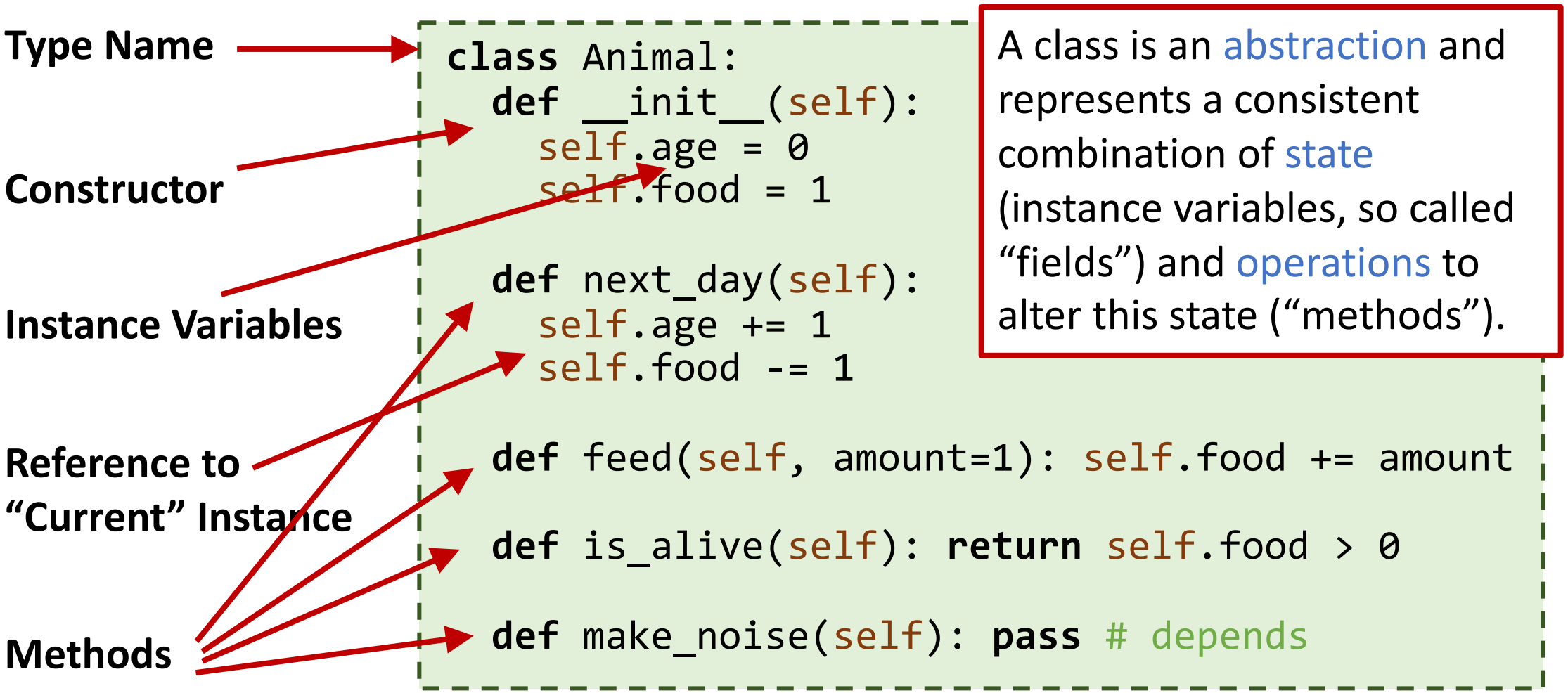
Eye Color

Hair Color

Facial Shape

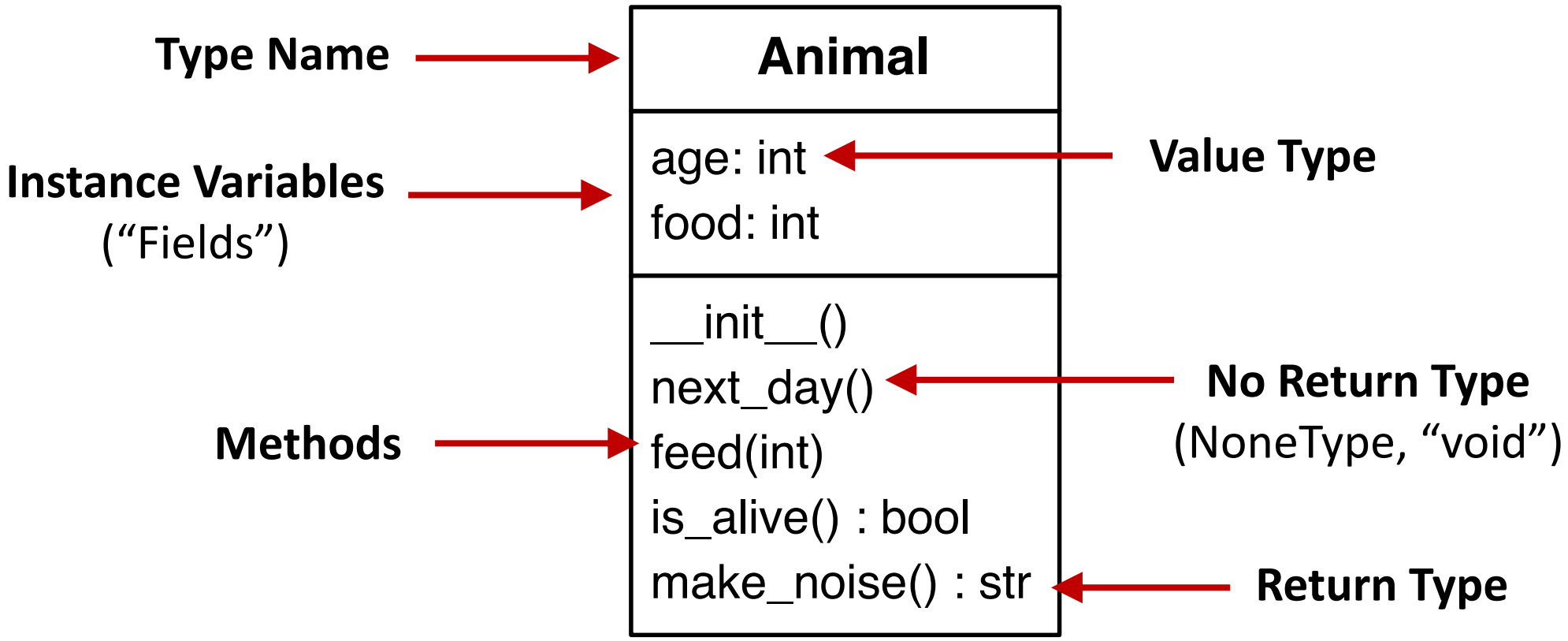
...

Animal Class



A class is an abstraction and represents a consistent combination of state (instance variables, so called "fields") and operations to alter this state ("methods").

Unified Modelling Language (UML)



Adding Concrete Animals

```
class Cat(Animal):
    def make_noise(self):
        return "Purrr"
```

Base Class

```
class Dog(Animal):
    def make_noise(self):
        return "Woof"
```

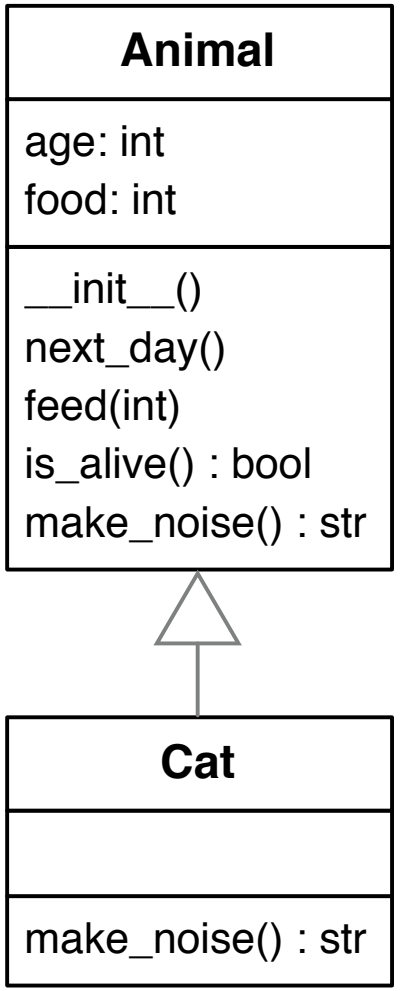
"Overridden" Method

The concrete class has all functionality of the base class

```
c = Cat()
c.feed()
print(c.is_alive()) # True
c.next_day()
print(c.is_alive) # True
c.next_day()
print(c.is_alive) # False
```

In OOP languages, classes can extend existing classes to inherit their behavior. At the same time, they can change and extend the behavior.

UML: Inheritance



Animal is the super class of **Cat**
Animal is the base class of **Cat**
Cat is a sub class of **Animal**
Cat is an **Animal**
Cat() is an instance of **Animal**
Cat and **Animal** are a type hierarchy.

make_noise is dynamically “dispatched”

OOP languages support Polymorphism. They can decide at the runtime of a program, depending on the type of the receiver object, which implementation of a specific method needs to be invoked.

```
Cat().make_noise() # Purrr
Dog().make_noise() # Woof
Animal().make_noise() # None
```

It does not make sense to instantiate “Animal”, how can we prevent it?

Abstract Base Class

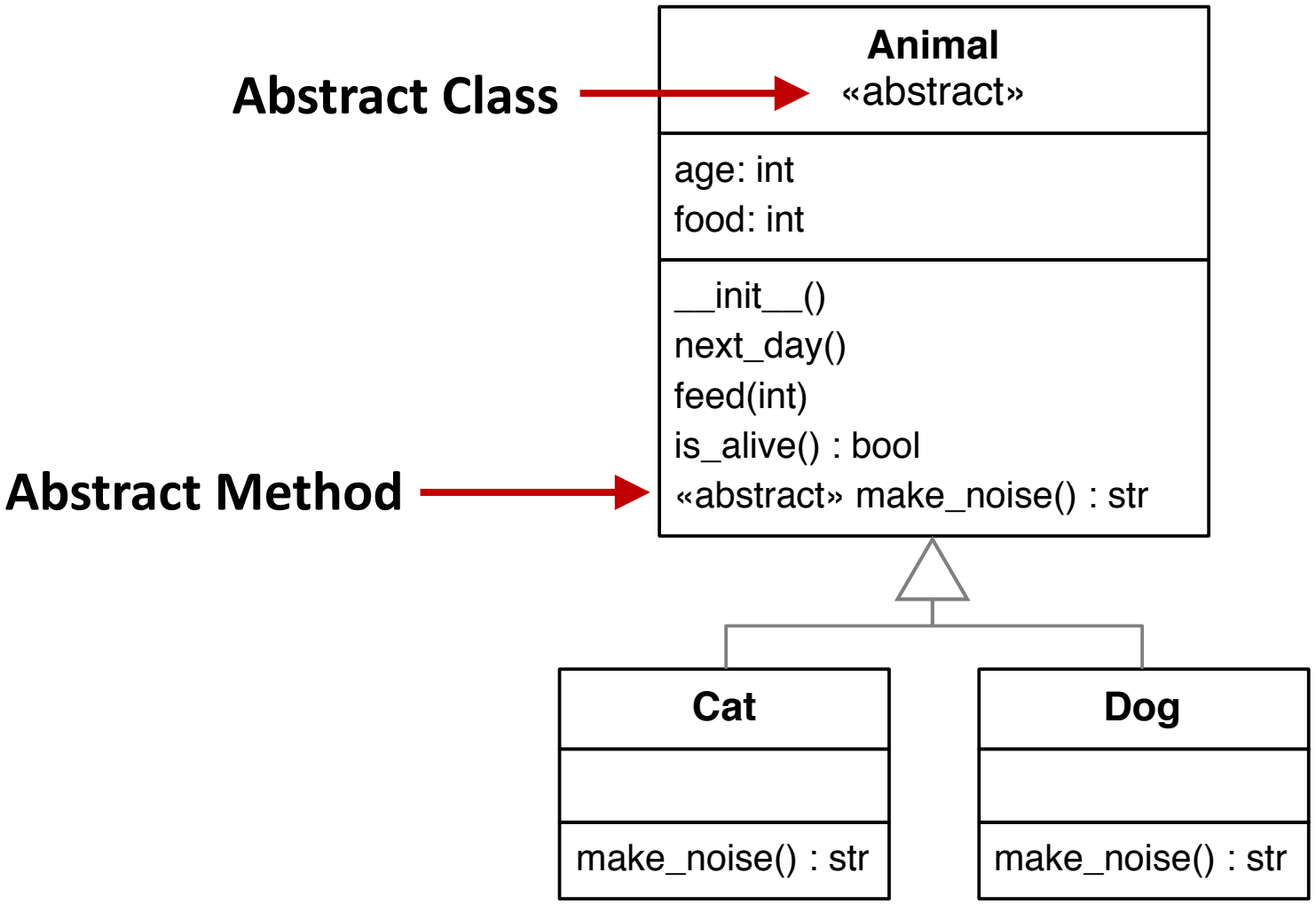
```
from abc import ABC, abstractmethod

class Animal(ABC): # ”A”bstract “B”ase “C”lass
  ...
  @abstractmethod
  def make_noise(self):
    pass
```

```
a = Animal()
# TypeError: Can't instantiate abstract class
Animal with abstract methods make_noise
```

An abstract base class can provide functionality that is relevant for subclasses without being constructable.

Define abstract methods in an abstract base class to indicate that subclasses must implement a particular method (template methods).

UML: Abstract



Elephants Need More Food

```
class Elephant(Animal):

  def next_day(self):
    self.food -= 3

  def make_noise(self):
    return “Toot”
```

```
class Animal(ABC):
  ...
  def next_day(self):
    self.__food -= self._hunger()
  def _hunger(self):
    return 1
```

Private

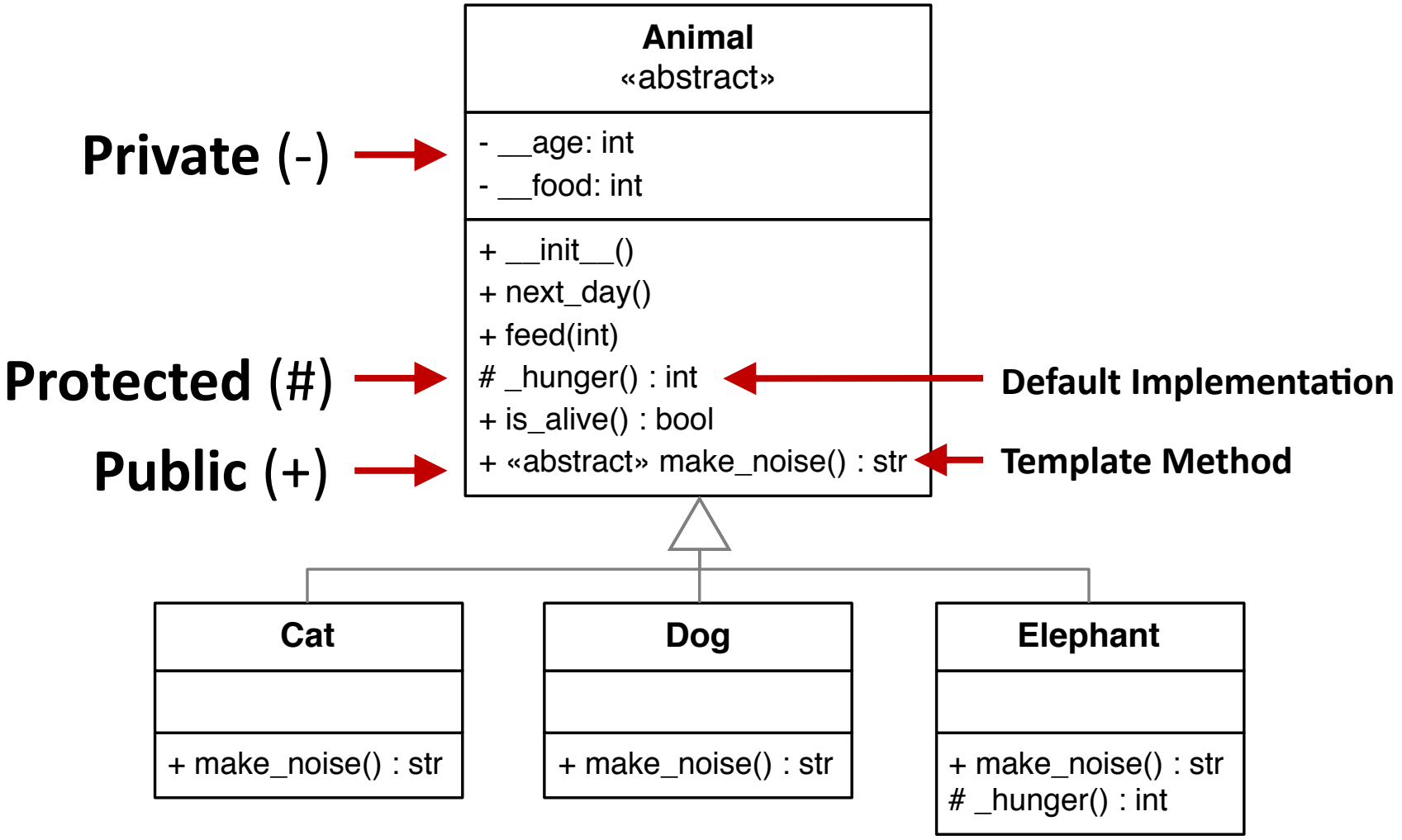
Protected

```
class Elephant(Animal):
  def _hunger(self):
    return 3
  def make_noise(self):
    return “Toot”
```

Public

Use visibility modifiers to hide implementation details. In contrast to public variables, private ones are visible in the class, protected ones are visible to the hierarchy.

UML: Visibility



What is the advantage of OOP?

```
zoo = [Cat(), Dog(), Elephant()]

for animal in zoo:
  assert isinstance(animal, Animal)

  # next day
  animal.next_day()

  # feed it
  if animal.is_alive():
    animal.feed(1)
  else:
    zoo.remove(animal)
```

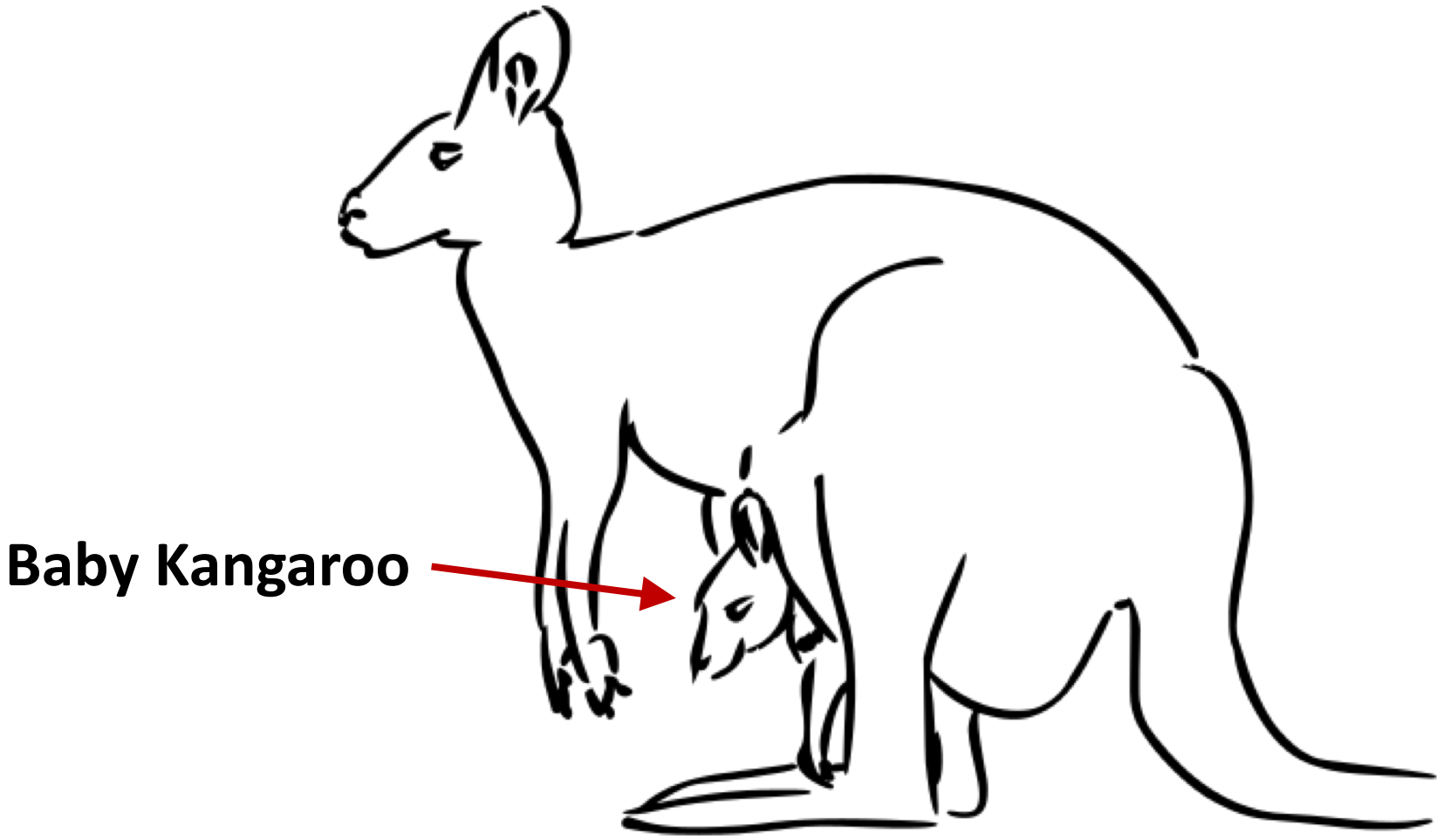
Often, you do not need the exact type of an object, it is enough to know the interface of an abstraction. This allows hiding implementation details of subtypes that add features.

```
if isinstance(animal, Elephant):
  # maybe “hunger” should be public
  animal.feed(3)
```

⚠ Little Word of Caution

- You CAN check for the concrete type, i.e., isinstance(o, «Type»)
- We are in an *Introduction to Programming* course, so this is ok!
- However, adding checks for concrete types breaks extensibility!
- If it is strictly required, it is a *smell* of a bad system design.
- More advanced techniques for type checking exceed the scope the lecture; if you are interested have a look at the Visitor design pattern.

Kangaroos Have Pouches



https://svgsilh.com/image/47642.html

Let's Implement Kangaroos (Take 1)

```
class Kangaroo(Animal):
  def __init__(self):
    self.__pouch = []
  def reproduce(self):
    self.__pouch.append(Kangaroo())
  ...
```


Now we have a pouch**, but what about** age **and** food**?**

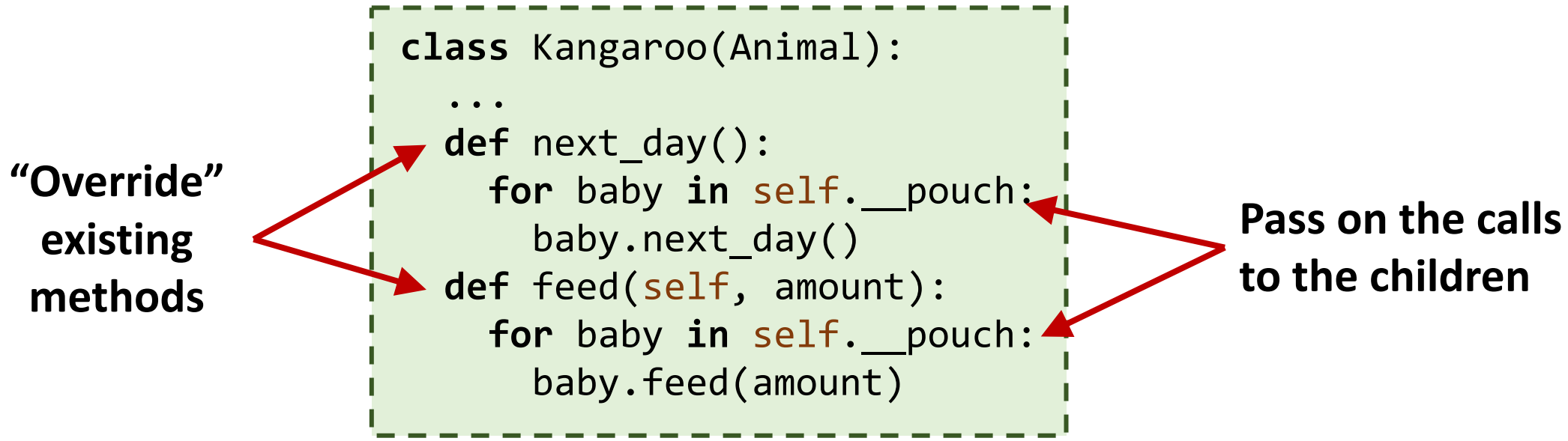
Let's Implement Kangaroo (Take 2)

```
class Kangaroo(Animal):
  def __init__(self):
    super().__init__()
    self.__pouch = []
  def reproduce(self):
    self.__pouch.append(Kangaroo())
  ...
```


Use `super()` to redirect method calls to the implementation in the super class

How do the baby kangaroos get fed? How do they age?

Let's Implement Kangaroo (Take 3)

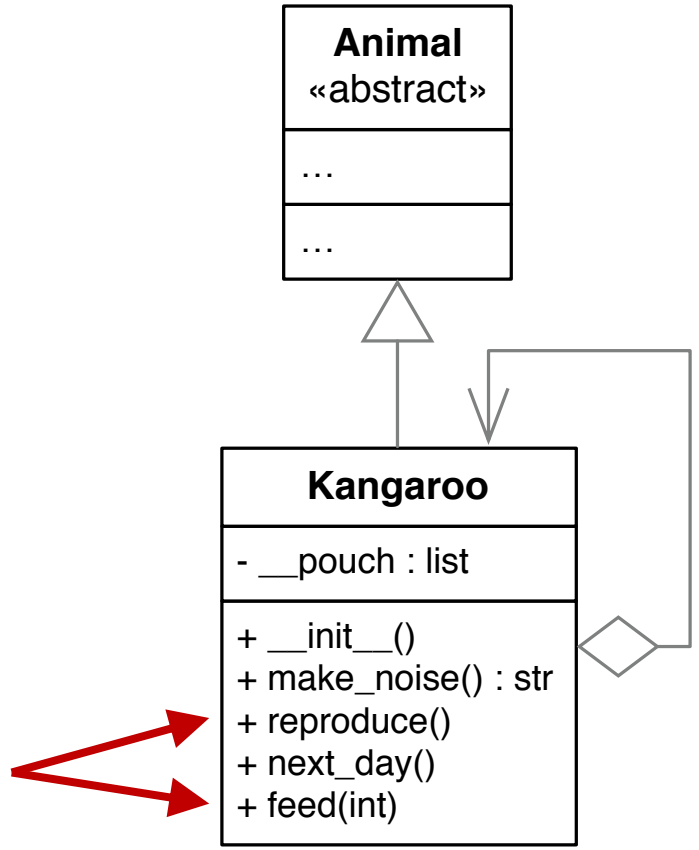


How does the mother get fed? How does she age?

Let’s Implement Kangaroo (Final)

```
class Kangaroo(Animal):
  ...
  def next_day():
    super().next_day()
    for baby in self.__pouch:
      baby.next_day()

  def feed(self, amount):
    super().feed(amount)
    for baby in self.__pouch:
      baby.feed(amount)
```

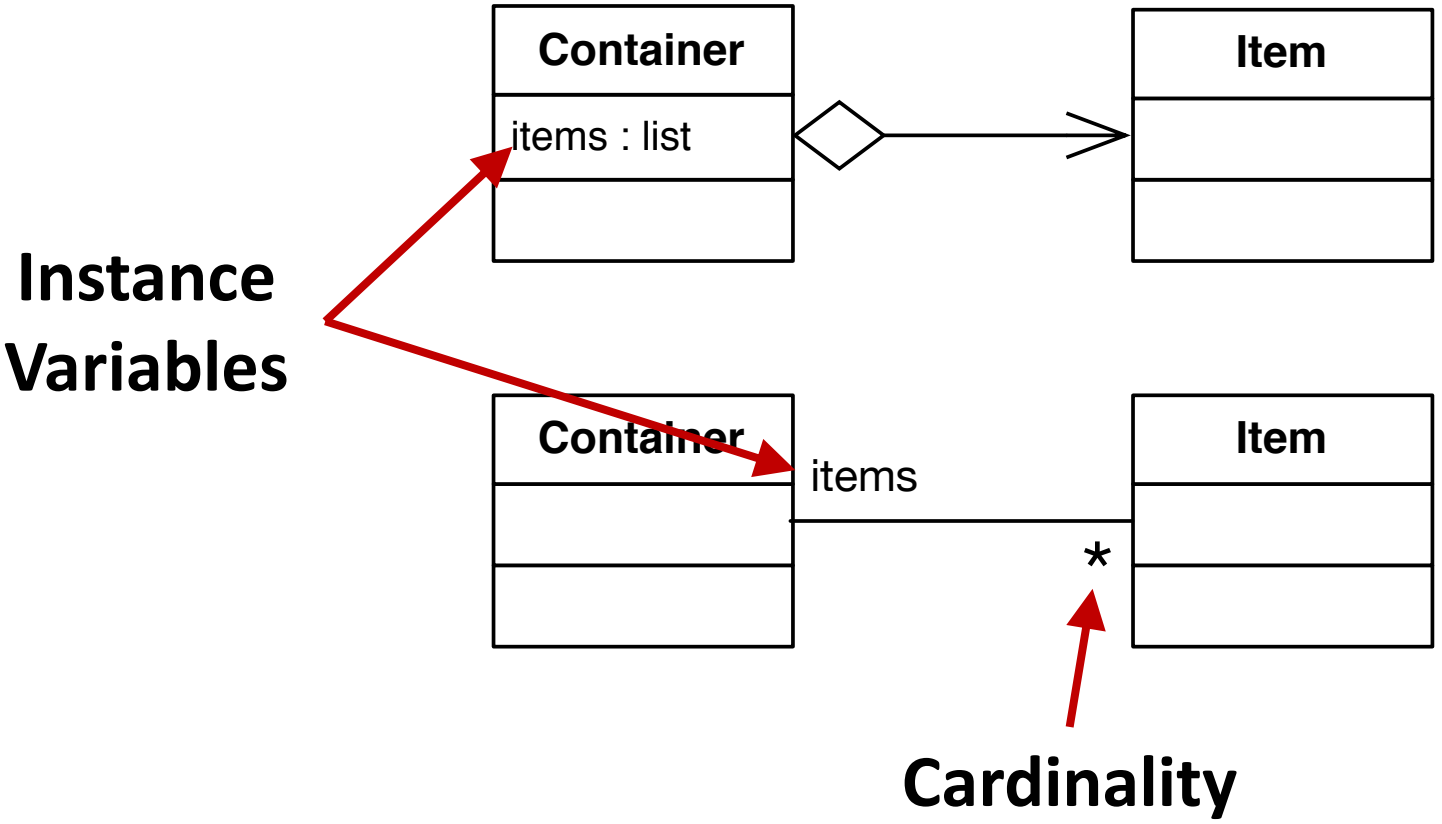
UML: Composition



Subclasses can replace existing methods and also add new ones

Kangaroo is a composite for other Kangaroos

UML: Composition and Cardinality



Exactly Once: 1
One or More: 1+
Optional: 1?
Bounded: 1..10
Unbounded: * (0..n)

The Safe, Revisited

https://www.allaboutsafes.co.uk/safes-c128/chubbsafes-elements-air-hotel-safe-p112

Requirements for a Safe Implementation

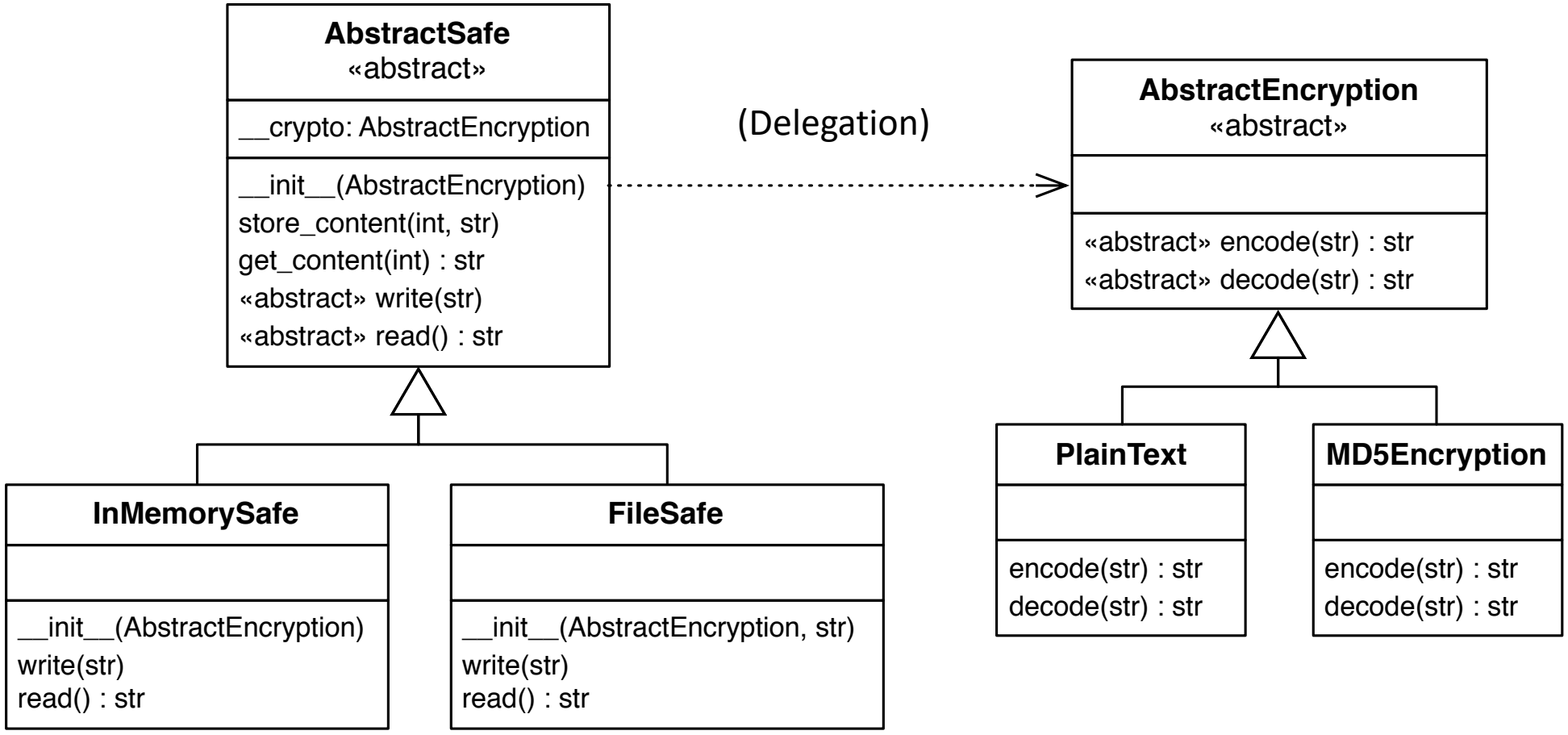
- Two Safe Variants
  - InMemorySafe
  - FileBasedSafe
- Two Encryptions
  - PlainText
  - MD5Encryption

```
a = InMemorySafe(PlainText())
b = InMemorySafe(MD5Encryption())
c = FileBasedSafe(PlainText(), “out.sav”)
d = FileBasedSafe(MD5Encryption(), “out.sav”)
```

```
def use_safe(safe):
    assert isinstance(safe, AbstractSafe)
    safe.store_content(123, “secret”)
    secret = safe.get_content(123)
```

Using abstractions allows to freely mix and match concrete implementations. Users don’t need to know.

UML: Designing a Safe



Advanced: Multiple Inheritance

Date and Time

```
class Time:
  def __init__(self, h, m, s): ...
  def add_hour(self): ...
  def add_minute(self): ...
  def add_second(self):
    self.__sec += 1
    if self.__sec > 59:
      self.add_min()
    self.__sec %= 60
  def __repr__(self):
    return "{:02}:{:02}:{:02}".format( \
      self.__hour, self.__min, self.__sec)
```

```
t = Time(1, 2, 3)
print(t) # 01:02:03
for i in range(70):
  t.add_second()
print(t) # 01:03:13
```

```
class Date:
  def add_year(self):
    ...
  def add_month(self):
    ...
  def add_day(self):
    ...
```

Multiple Inheritance

```
class DateTime(Date, Time):

  def __init__(self, d,m,y, h,min,s):
    Date.__init__(self, d, m, y)
    Time.__init__(self, h, min, s)

  def __repr__(self):
    d = Date.__repr__(self)
    t = Time.__repr__(self)
    return "{}-{}".format(d, t)

  def add_hour(self):
    before = self._hour
    Time.add_hour(self)
    if before > self._hour:
      self.add_day()
```

```
t = DateTime(1, 2, 3, 4, 5, 6)
print(t) # 01.02.0003-04:05:06
for i in range(25)
  t.add_hour()
print(t) # 02.02.0003-05:05:06
```

Using multiple inheritance, a class inherits the behavior of multiple parents. Conflicts are resolved by parent order, ”super” needs to be explicit.

Mix-In

```
class TimeMixin:
  def add_several_hours(self, amount):
    for i in range(amount):
      self.add_hour()

tm = TimeMixin()
tm.add_several_hours()
# AttributeError: 'TimeMixin' object
has no attribute 'add_hour'
```

```
class MyTime(Time, TimeMixin): pass

t = MyTime(1, 2, 3)
print(t) # 01:02:03
t.add_several_hours(10)
print(t) # 11:02:03
```

Using mixins, it is possible to change behavior of a class that does not belong to the same type hierarchy.

Summary

The Four Core Principles of OOP

- **Abstraction**
  - Define a concept, unrelated to a concrete instance.
- **Encapsulation**
  - Hide implementation details (Fields cannot be overridden!).
- **Inheritance**
  - Extend classes to reuse code (inherit behavior), *is-a* relationships.
- **Polymorphism**
  - Replacing functionality in specializations. Dynamic selection of methods.

You should be able to answer these questions

- What is the idea behind inheritance? What is a type hierarchy?
- How to define (abstract) base classes? How to extend them?
- Which visibility levels exist and how do you declare them?
- How to provide classes that act as a composite for other items.
- How to delegate sub-problems to other referenced classes.
- How to read and write OOP designs in Unified Modelling Language
- How to use multiple inheritance and mixins in Python?
- OOP Core Principles: Polymorphism/Information Hiding

Exercise

Implement the “Safe” Hierarchy

- Left as an exercise for the reader.