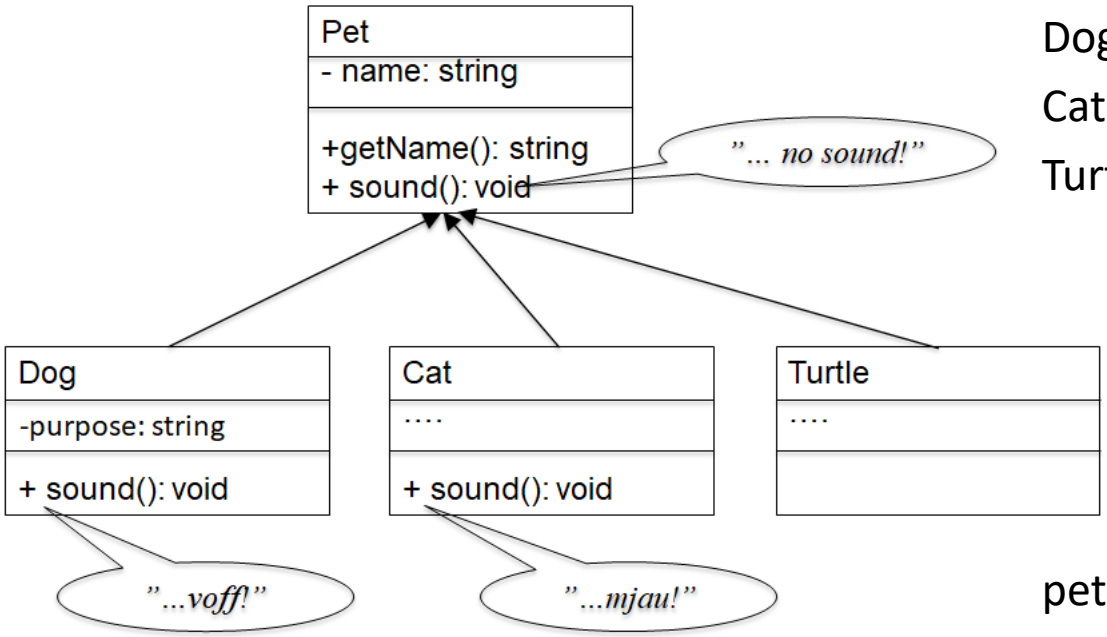

# BTH001
# Object Oriented Programming
# Lesson 05
# Inheritance continued

# Function overriding

- It is possible to **override (redefine)** functions in an inheritance hierachy

- If a function is overridden and an object makes a call of the overridden function the definition that is consistent with the declaration of the variable is used. This is **static (early) binding**

# Pet example

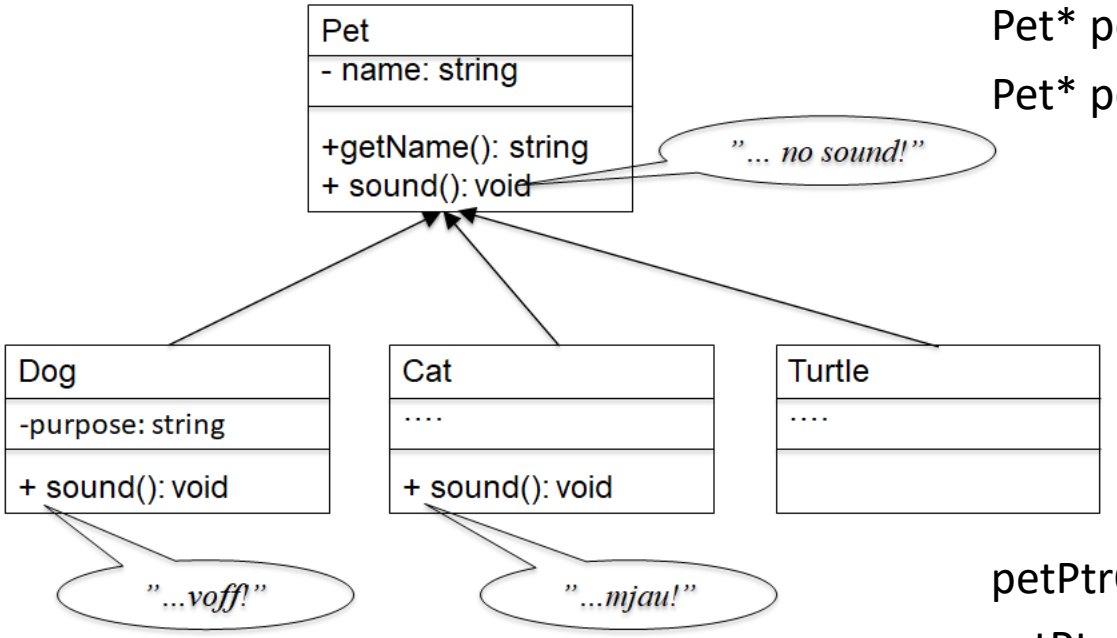



```
Pet petObject(”The Pet”);
Dog dogObject(”Lady”, ”Hunting”);
Cat catObject(”Duchess);
Turtle turtleObject (”Shell”);
```

```
petObject.sound();
dogObject.sound();
catObject.sound();
turtleObject.sound();
```

# Pet example (cont)

```
Pet* petPtrOne = new Pet(”The Pet”);
Cat* catPtr = new Cat(”Duchess”);
Pet* petPtrTwo =  new Dog(”Fido”,”Social”);
Pet* petPtrThree = new Turtle(”Shell”);
```

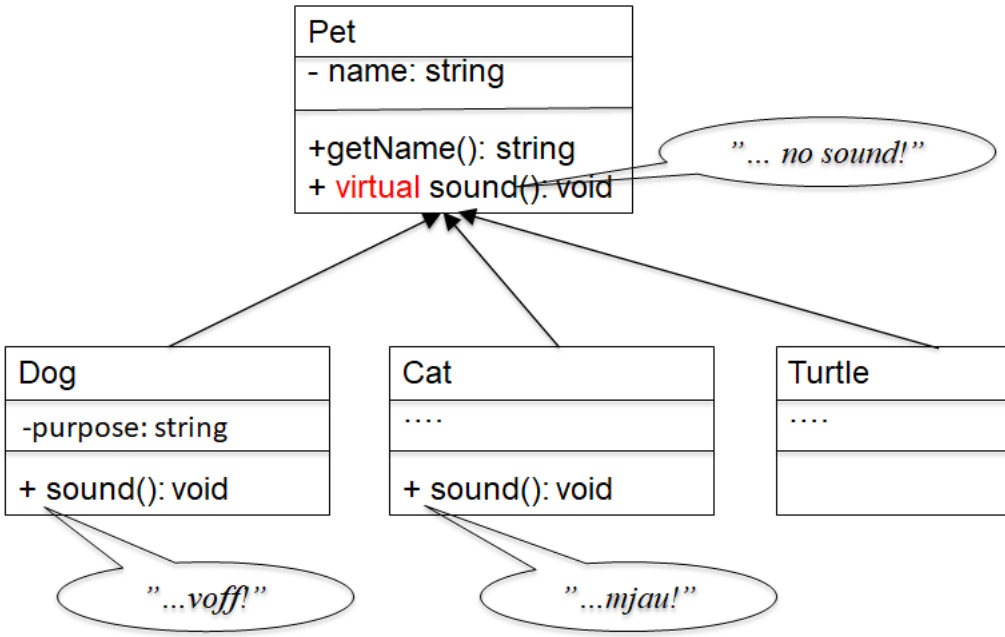



```
petPtrOne->sound();
catPtr->sound();
petPtrTwo->sound();
petPtrThree->sound();
```

# Virtual functions

- An overridden function that is declared virtual in the base class will enable **dynamic bindning (late binding)**
- Dynamic (late) binding means that it is the **type of the object** (not the type of the variable) that decides which definition of the function that will be used. Only relevant when using **pointers of base class type.**

# Pet example (cont)




```
Pet* petPtrOne = new Pet("The Pet");
Cat* catPtr = new Cat("Duchess");
Pet* petPtrTwo =  new Dog("Fido", "Social"
Pet* petPtrThree = new Turtle("Shell");
```

```
petPtrOne->sound();
catPtr->sound();
petPtrTwo->sound();
petPtrThree->sound();
```

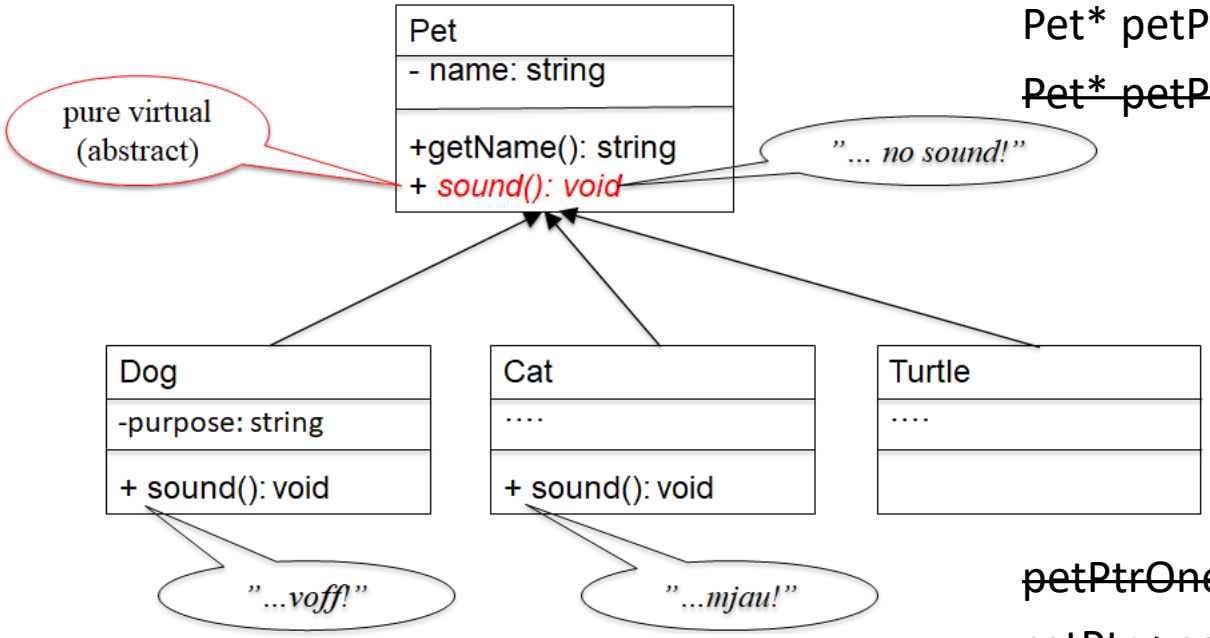
# Pure virtual functions

- A **pure virtual function** is a function that requires a definition in the derived class, that means that the function must be overridden
- If a class contains one or more pure virtual functions the class become **abstract**
- If a pure virtual function isn't overridden in the derived class the derived class will become abstract

# Abstract classes

- Base classes can be abstract
- Abstract classes can **not be instansiated**
- Abstract classes can be used as a type. Variables can be declared as pointers of abstract class type
  - BaseClass *variable;

# Pet example (cont)




~~Pet* petPtrOne = new Pet("The Pet");~~

Cat* catPtr = new Cat("Duchess");

Pet* petPtrTwo = new Dog("Fido", "Social");

~~Pet* petPtrThree = new Turtle("Shell");~~

~~petPtrOne->sound();~~

catPtr->sound();

petPtrTwo->sound();

~~petPtrThree->sound();~~

# Abstract classes in C++

```
// header file of abstract class
class A
{
    // members
   virtual void print() = 0;
};
```

```
// header-file of concrete class
class B: public A
{
    // members
   void print();
};
```

```
// cpp-file of concrete class
void class B::print()
{
   cout<<"B ";
}
```

# Abstract, virtual and pure

- A **virtual function enables dynamic (late) bindning** instead of static (early) bindning
- **Pure virtual functions make a class abstract**
- A **pure virtual function requires a definition in derived classes**, otherwise the derived classes will become abstract as well
- An **abstract class can not be instantiated**
- **Pointer variables can be declared as abstract class type**

# Virtual destructor

- To accomplish late **binding of destructors**, the destructors has to be **virtual**
- This is relevant when pointers of base class type addresses (points to) objects of sub class type.

# Dynamic_cast

- When you need to treat an object as its specific sub class type instead of its base class type you need to
  - identify the type of the object (the dynamic type)
  - use type casting

# Decide the type of an object

In C++

- dynamic_cast<type *>(pointer to the object);

Example:

- Person *pers = nullptr;
- pers is assigned the adress to a Student or an Employee object

- if (dynamic_cast<Student *>(pers) != nullptr) {…}