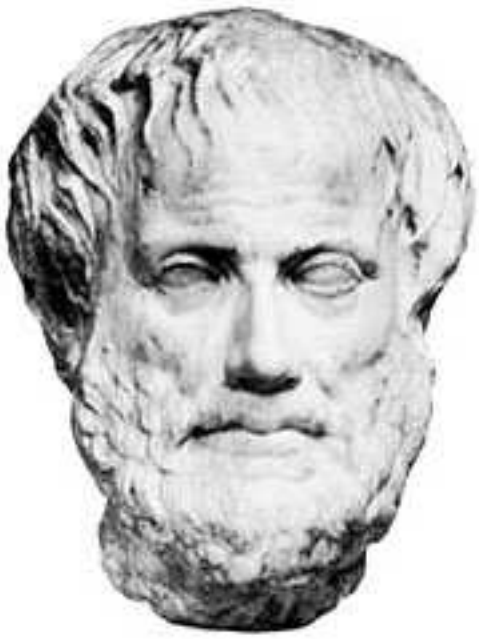
Review

- Sorting arrays of objects:
  - Objects must have a key attribute
  - Keys must be comparable (their class must provide a compareTo method)
- Multi-dimensional arrays
- Memoization: using arrays to optimize recursive methods by keeping track of previously computed solutions.

Object oriented Programming

- The execution of an OO program consists of
  - Creation of objects
  - Interaction between objects (message-passing)
- Defining features of an OO language:
  - Class definitions (describing the types of objects and their structure,)
  - Objects made up of attributes and methods (as given by the object's class,)
  - Object instantiation (creation,)
  - Message-passing (invoking methods,)
  - Encapsulation (objects as abstract units, hiding,)
  - Inheritance,
  - Polymorphism

Inheritance



- Being of some kind

Inheritance

- Aristotle's silogism describing aggregation (structure of objects)
  - If every A ***has a*** B and x ***is an*** A then x ***has a*** B
    - (e.g. If every dog has a tail and Grommit is a dog, then Grommit has a tail)
  - If every A ***can do an action*** P and x ***is an*** A then x ***can do an action*** P
    - (e.g. if every dog can bark, and Grommit is a dog, then Grommit can bark)

```
class Dog {
  Tail t;
  void bark() { ... }
}
// Somewhere else...
Dog grommit = new Dog();
grommit.bark();
... grommit.t ...
```

Inheritance

- Aristotle's silogisms describing inheritance
  - if every A ***is a*** B and x ***is an*** A then x ***is a*** B
    - (e.g. if every labrador is a dog and Grommit is a labrador then Grommit is a dog)
  - if every A ***is a*** B and every B ***has a*** C then every A ***has a*** C
    - (e.g. if every dog has a tail and every labrador is a dog then every labrador has a tail)

Inheritance

- Two "kinds" of "is-a" relationship:
  - Between an individual (object) and its class (x is of type A, e.g. Tokyo is a City)
  - Between two classes (every A is a B, e.g. every dog is a mammal.)

- In the first silogism, when we say "x ***is an*** A", x is an individual, we are talking about a specific x who is a kind of A, in other words, x is an object, and A is a class of objects, so x is an instance of class A.

- In the second silogism, when we say "every A ***is a*** B", we are talking about all A's, all individuals who are A's. This is equivalent to saying:
  - "for all individuals x, if x *is an* A, then x *is also a* B."
  - ... or, "for all objects x, if x is of type A, x is also of type B."

Inheritance

- The first kind represents instantiation
- The second, represents inheritance
- Representing the two kinds of "is-a" in Java:
  - Between an individual (object) and its class (x is of type A, e.g. Tokyo is a City)

```
A x = new A();
City tokyo = new City();
```

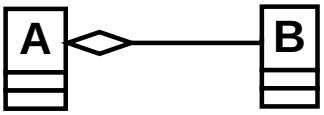
  - Between two classes (every A is a B, e.g. every dog is a mammal.)

```
class B { ... }
class A extends B { ... }
class Labrador extends Dog { ... }
```

- We say that A is a subclass of B, or A is derived from B, or B is a superclass of A, or B is a parent of A.

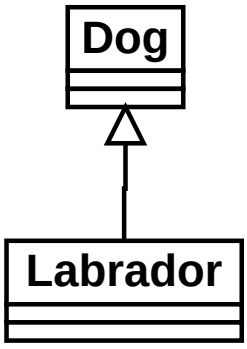
Inheritance

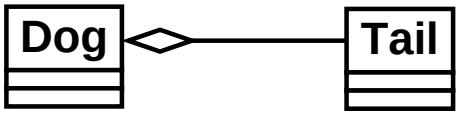
represents:
"every A is a B"
(inheritance)



represents:
"every A has a B"
(aggregation)

For example:





Inheritance

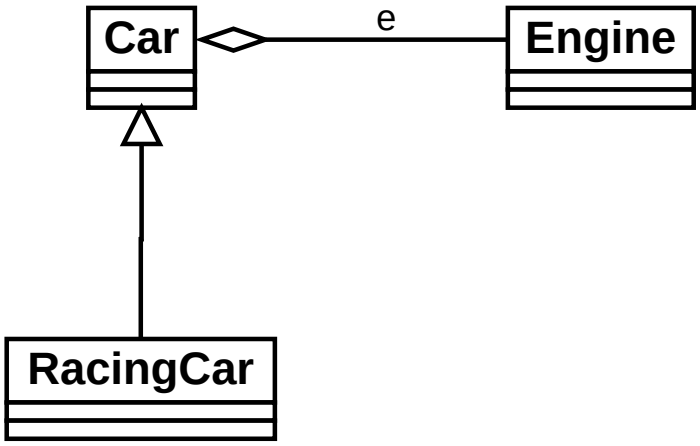
- The silogism "if every A ***is a*** B and every B ***has a*** C then every A ***has a*** C", means that all the attributes that B has, are also attributes of A. A may have other attributes as well which B doesn't. A is more specific or specialized than B.

```
class C { ... }
class B {
  C v;
  // ...
}
class A extends B {
  // Has an implicit C v;
  // ...
}
```

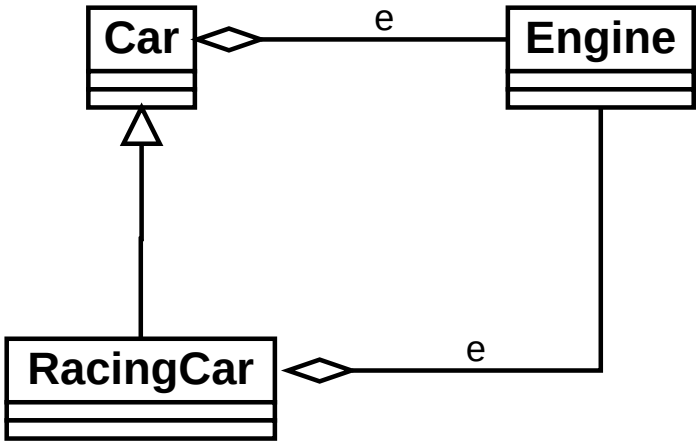
Inheritance

```
class Engine {
  // ...
}

class Car {
  Engine e;
  // ...
}

class RacingCar extends Car {
  // It implicitly has Engine e;
  // ...
}

// In some client
RacingCar r = new RacingCar();
Engine e = r.e;  // e is inherited from Car
```

Inheritance



is the same as



Inheritance

- Inheritance also represents specialization

```
class Engine {
  // ...
}
class Car {
  Engine e;
  Car() { e = new Engine(); }
  // ...
}
class RacingCar extends Car {
  Aerofoil a;
  TurboCharger t;
}

// In some client
RacingCar r = new RacingCar();
Engine e1 = r.e;  // e is inherited from Car
TurboCharger t1 = r.t;
Car c = new Car();
Engine e2 = c.e;
TurboCharger t2 = c.t; // Error
```

Inheritance

- Inheritance serves as a tool for reusability:
- We can write

```
class RacingCar extends Car {
  Aerofoil a;
  TurboCharger t;
}
```

instead of

```
class RacingCar {
  Engine e;
  Aerofoil a;
  TurboCharger t;
}
```

Inheritance

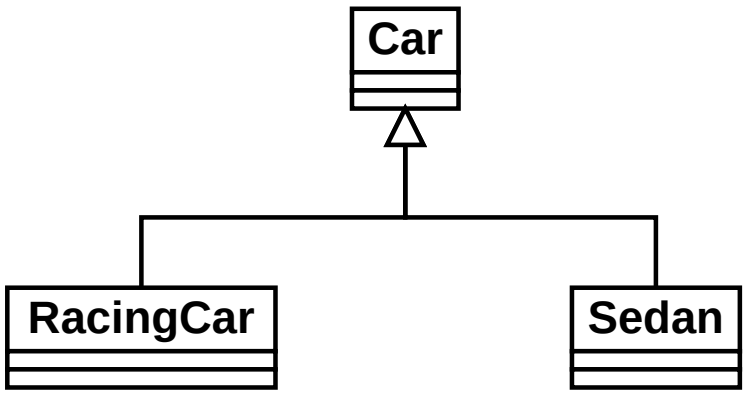
- Methods are inherited too:

```
class Engine {
  void start() { ... }
}
class Car {
  Engine e;
  double speed;
  Car() { e = new Engine(); speed = 0.0; }
  void turn_on()
  {
    e.start();
  }
}
class RacingCar extends Car {
  Aerofoil a;
  TurboCharger t;
}
// In some client
RacingCar r = new RacingCar();
r.turn_on(); // Inherited from Car
```

Inheritance

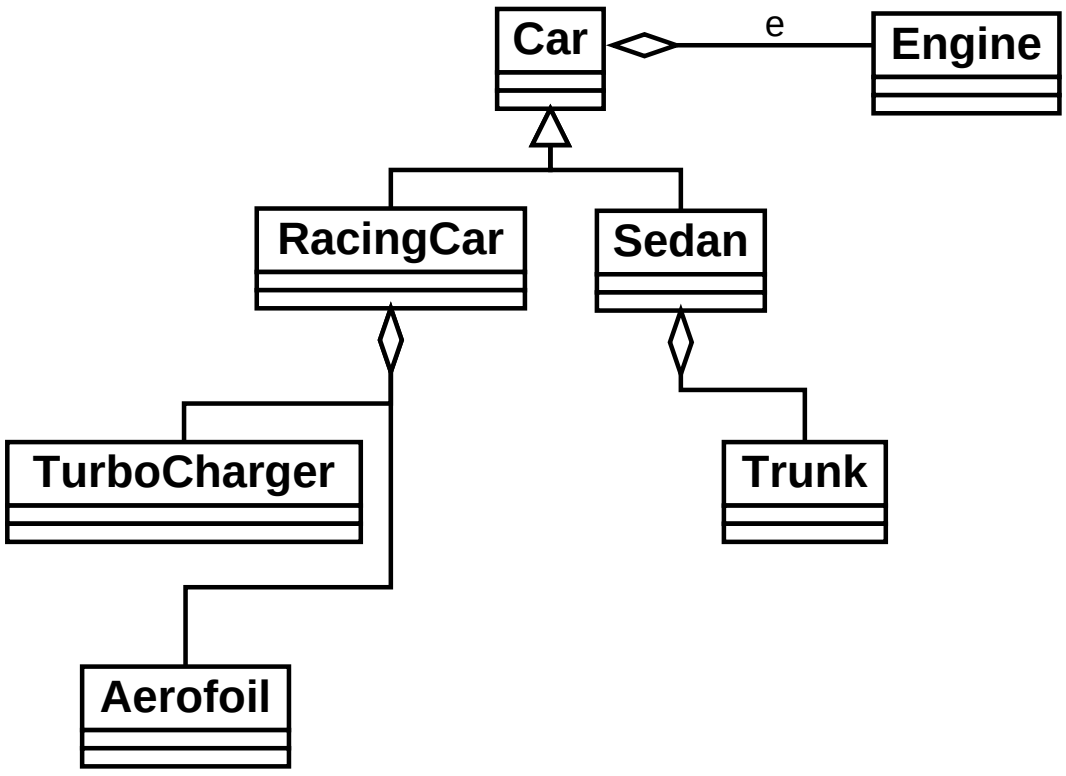
- Classes can have many subclasses

```
class Sedan extends Car {
  Trunk t;
  PassengerSeats[] ps;
}
// In some client
Sedan s = new Sedan();
s.turn_on();
```



Inheritance

- Attributes in a class are shared between its subclasses (but not the values of those attributes!)



Inheritance

- Inheritance is a transitive relation: if every A is a B and every B is a C, then every A is a C

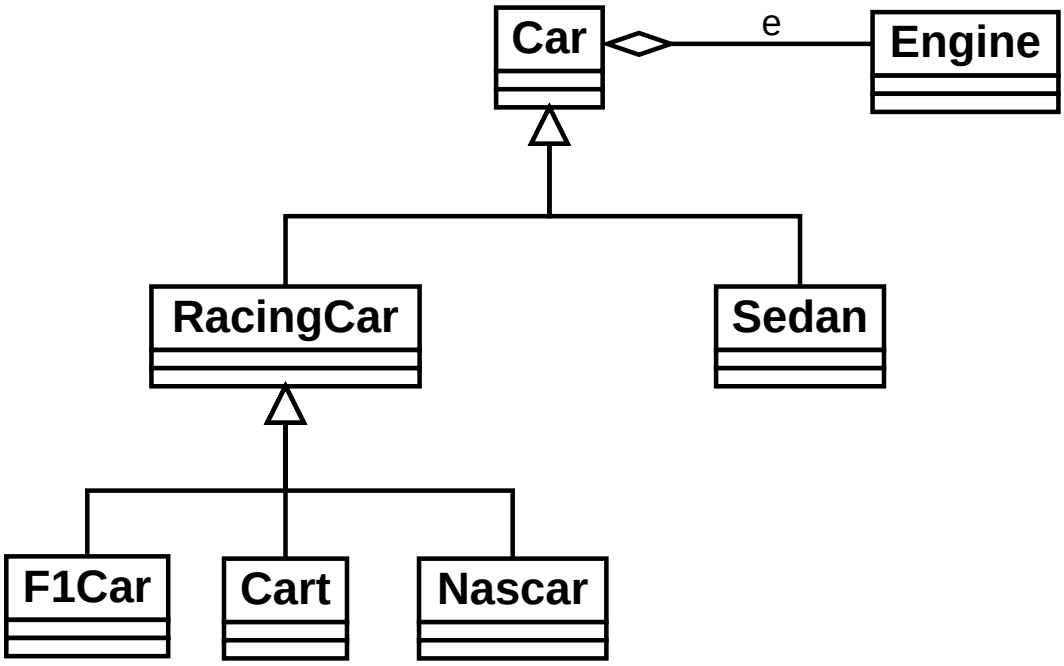
```
class F1Car extends RacingCar {
  SpeedControlSystem scs;
}
```

- instead of

```
class F1Car {
  Engine e;
  Aerofoil a;
  TurboCharger t;
  SpeedControlSystem scs;
}
```

Inheritance

- Class hierarchy:



Inheritance

- A closer look at inheritance as specialization

```
class Animal {
  boolean tired, hungry;
  void eat()
  {
    get_food();
    hungry = false;
  }
  void get_food() { ... }
  void sleep()
  {
    System.out.println(“zzz...”);
    tired = false;
  }
}
```

Inheritance

```
class Dog extends Animal {
  Legs[] l;
  Tail t;
  void run()
  {
    tired = true; // From class Animal
    hungry = true;
  }
  void bark()
  {
    System.out.println("Woof, Woof!");
  }
}
class Labrador extends Dog {
  void say_hello()
  {
    t.wiggle(); // t from class Dog
  }
}
```

Inheritance

```
public class ZooTest {
  public static void main(String[] args)
  {
    Labrador l = new Labrador();
    l.say_hello(); // Will call l.t.wiggle();
    l.run();
    if (l.hungry)
      l.eat(); // from class Animal
    if (l.tired)
      l.sleep();
  }
}
```

Inheritance

- Inheritance represents also a spectrum of possibilities or alternatives, given by the subclasses of a class
- If every B is an A and every C is an A, and nothing else is an A, then an A is either a B or a C
  - (e.g. if every racing car is a car, and every sedan is a car, and nothing else is a car, then a car is either a racing car or a sedan.)

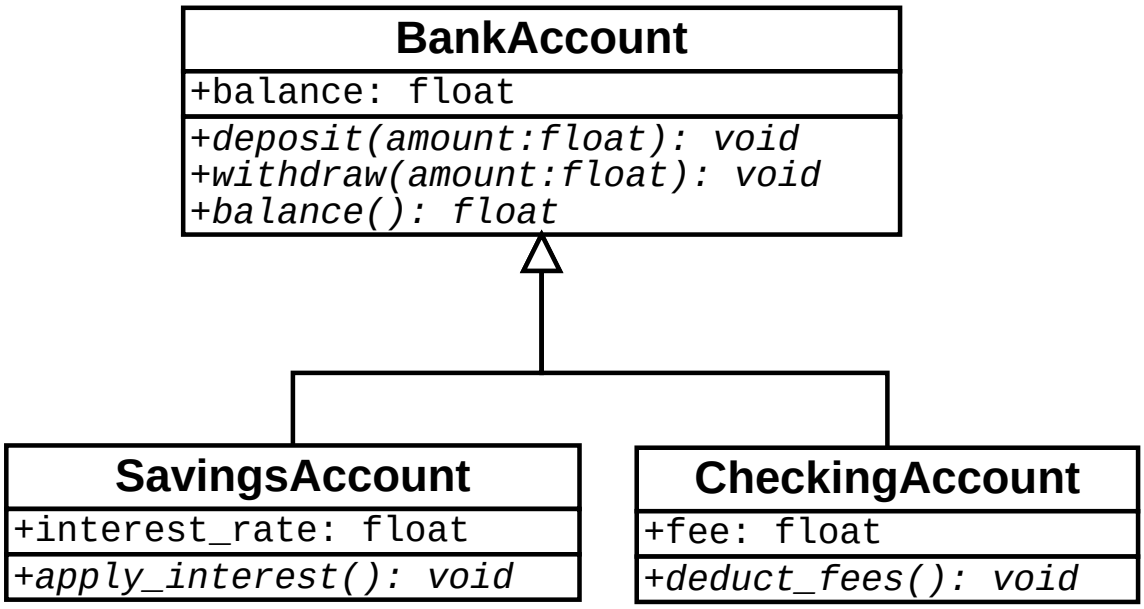
```
class Animal { ... }
class Dog extends Animal { ... }
class Cat extends Animal { ... }
class Bird extends Animal { ... }

// In some client
Animal a1 = new Dog();
Animal a2 = new Cat();
Animal a3 = new Bird();
Dog d = new Animal(); // Wrong!
```

Inheritance

- Classes as sets of objects:
  - "is-a" between an object and a class is the same as $\in$
  - "is-a" between two classes is the same as $\subseteq$
- Let A, B, C be sets
  - If $A \subseteq B$ and $x \in A$ then $x \in B$
  - If $A \subseteq B$ and $B \subseteq C$ then $A \subseteq B$
  - If $B \subseteq A$ and $C \subseteq A$, and there is no other set D such that $D \subseteq A$ then $A = B \cup C$

Inheritance

- A bank account is either a savings account or a checking account, then a savings account is a kind of bank account, and a checking account is a kind of bank account.



Inheritance

```
class BankAccount {
  private float balance;
  public BankAccount(float initial_balance)
  {
    balance = initial_balance;
  }
  public void deposit(float amount)
  {
    balance = balance + amount;
  }
  public void withdraw(float amount)
  {
    balance = balance - amount;
  }
  public float balance() { return balance; }
}
```

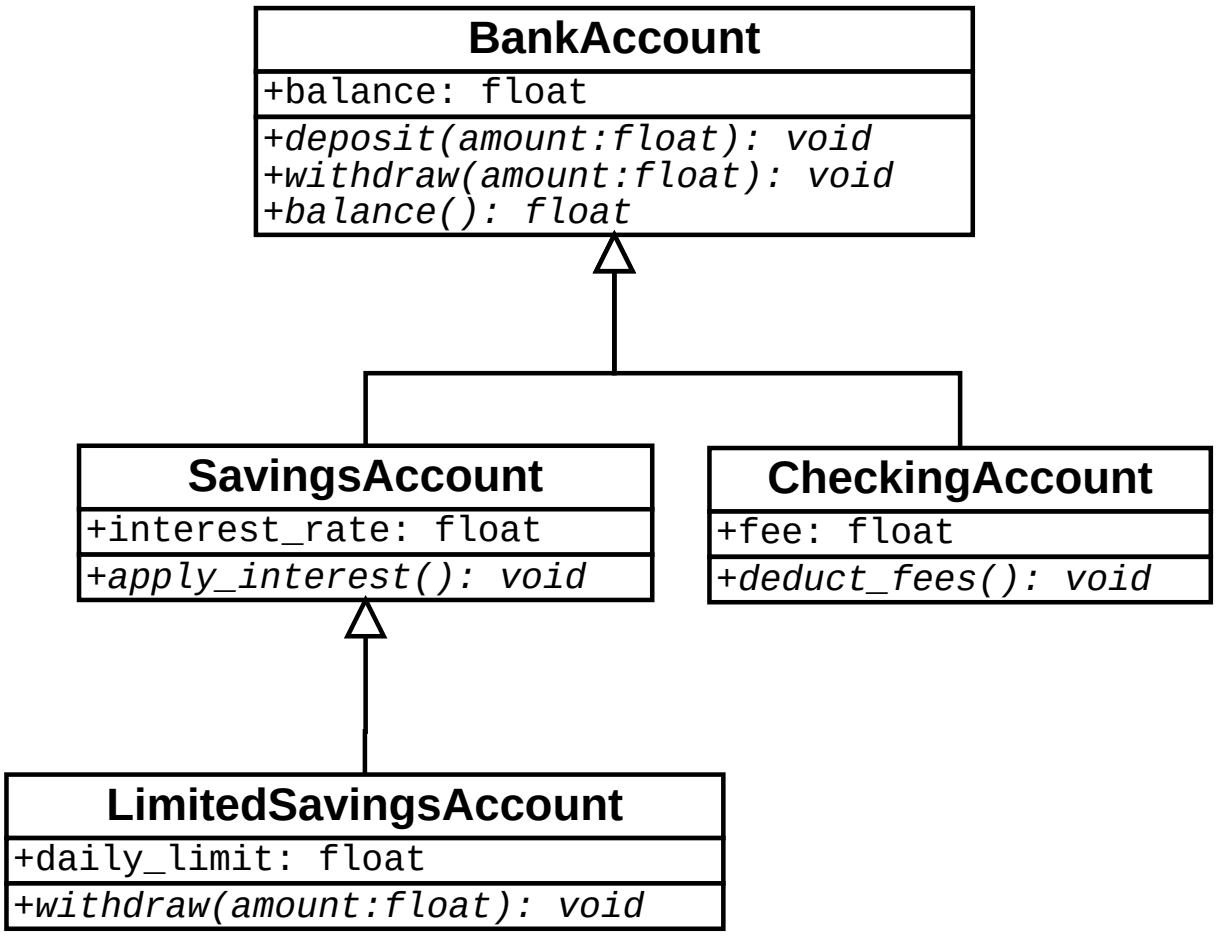
Inheritance

```
class SavingsAccount extends BankAccount {
  private float interest_rate;
  public SavingsAccount(float initial_balance,
                        float rate)
  {
    super(initial_balance); // Calls superclass
                            // constructor
    interest_rate = rate;
  }
  public void apply_interest()
  {
    balance = balance
            + balance * interest_rate/100.0;
  }
}
```

Inheritance

```
class CheckingAccount extends BankAccount {
  private float fee;
  public SavingsAccount(float initial_balance,
                        float fee)
  {
    super(initial_balance);
    this.fee = fee;
  }
  public void deduct_fee()
  {
    balance = balance - fee;
  }
}
```

Overriding methods



Overriding methods

```
class LimitedSavingsAccount
extends SavingsAccount {
  private float daily_limit;
  public LimitedAccount(float initial_balance,
                        float rate, float limit)
  {
    super(initial_balance, rate);
    daily_limit = limit;
  }
  public void withdraw(float amount)
  {
    if (amount < daily_limit)
      balance = balance - amount;
  }
}
```