Review

- Inheritance:
  - Represents the "is-a" relationship between classes
  - Represents specialization of classes (subsets)
  - Represents a way of describing alternatives (alternative subclasses)
  - Is a mechanism for reusability

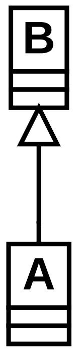
Inheritance

- Whenever we have a situation which states that "every A is a B", we model this as

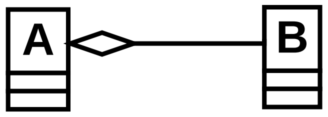
```
class A extends B { ... }
```

- All attributes and methods from the parent class (or super class) B are "inherited" by the subclass (or derived class) A.

- Class A can have (and usually does have) additional attributes and methods.



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



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

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 labrador is a dog and every dog has a tail then every labrador has a tail)

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

- 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
```

McGill

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:

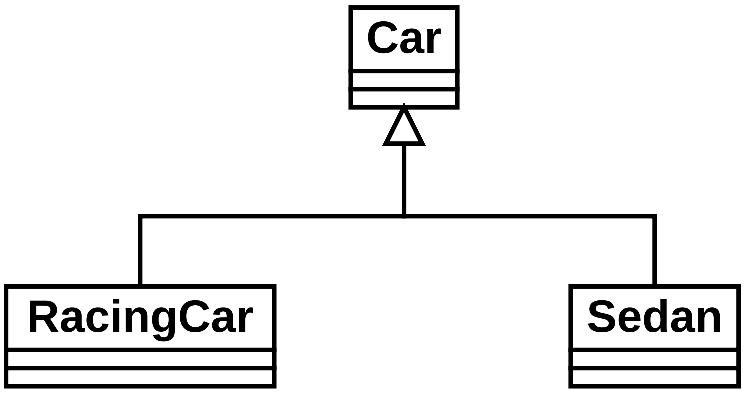
```
class Engine {
  void start() { ... }
}
class Car {
  Engine e;
  double speed;
  Car() { e = new Engine(); speed = 0.0; }
  void turn_on()
  {
    e.start();
  }
}
class RacingCar extends Car {
  Aerofoil a;
  TurboCharger t;
}
// In some client
RacingCar r = new RacingCar();
r.turn_on(); // Inherited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

McGill

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

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

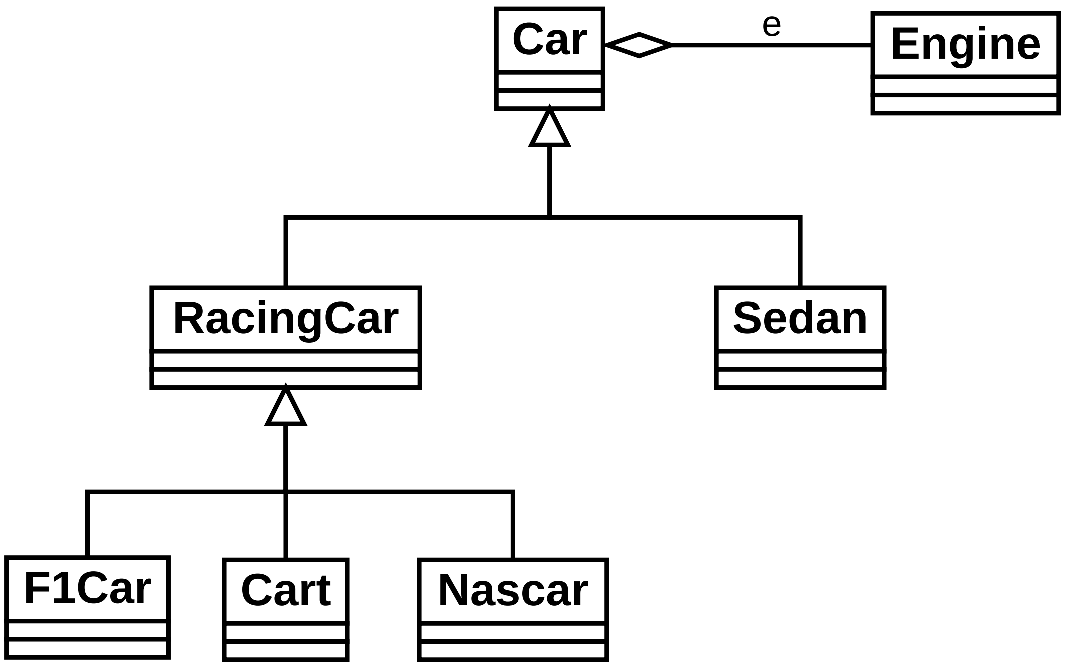
```
class F1Car extends RacingCar {
  SpeedControlSystem scs;
}
```

- 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.)

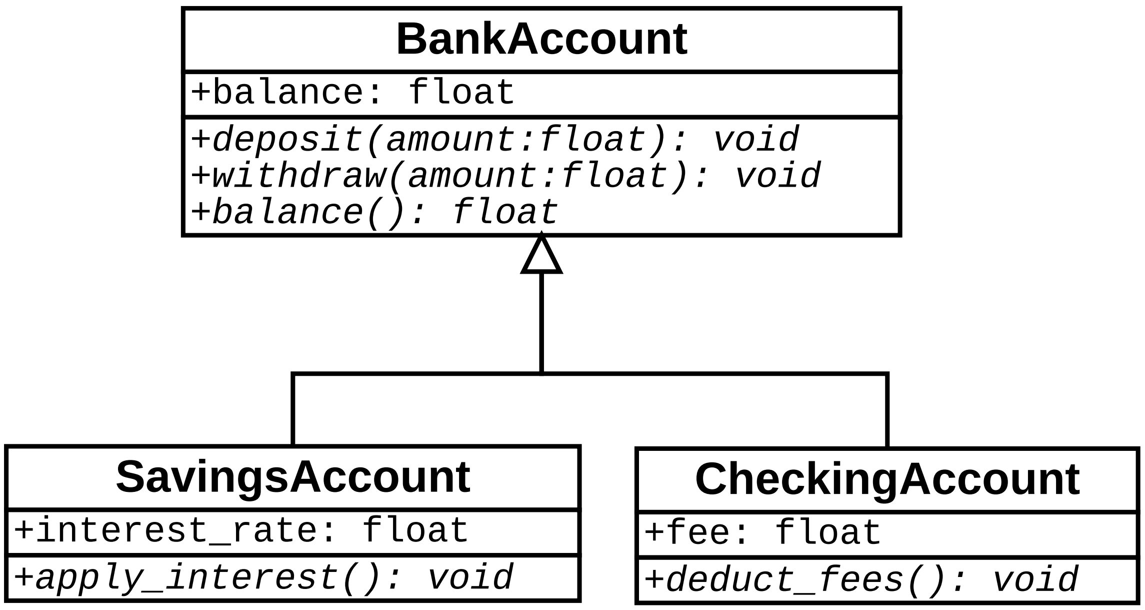
```
class Animal { ... }
class Dog extends Animal { ... }
class Cat extends Animal { ... }
class Bird extends Animal { ... }

// 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; }
}
```

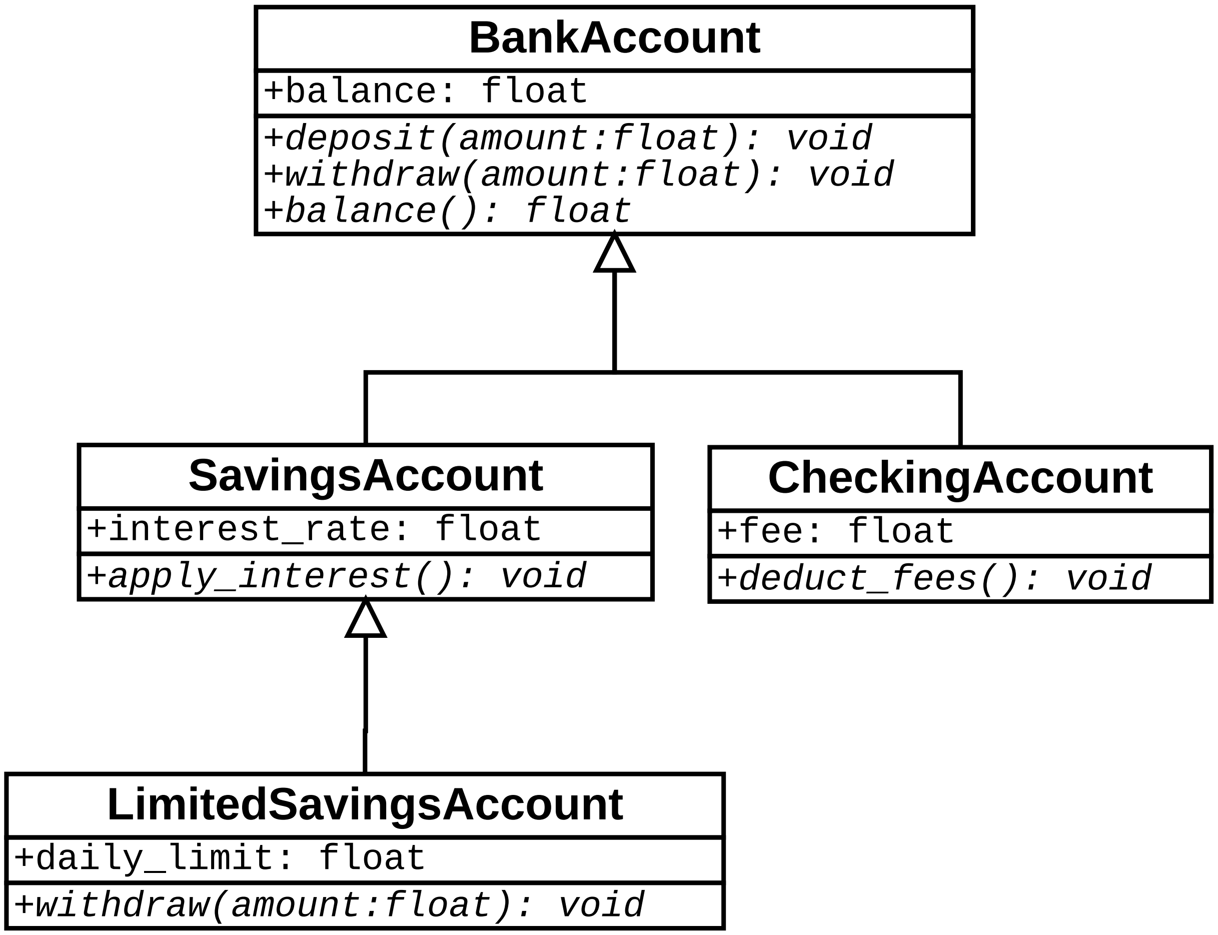
Inheritance

```
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;
  }
}
```

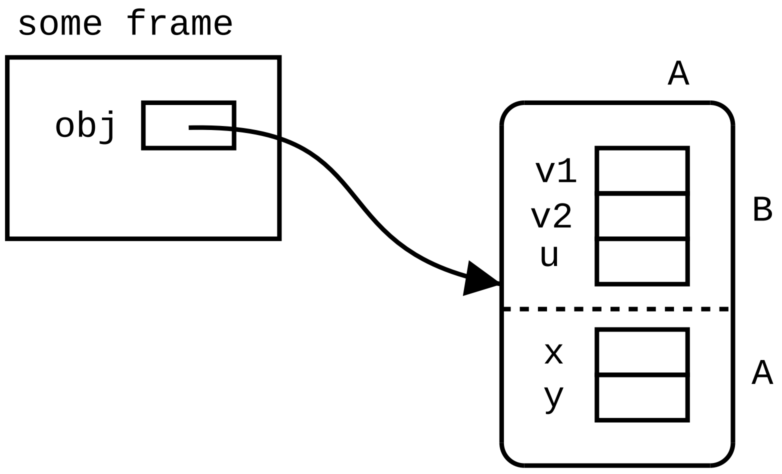
Inheritance

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... }
}
class A extends B {
  E x;
  C y;
  void p() { ... }
  void s() { ... }
}
```

McGill

Inheritance

```
// In some client
A obj = new A();
obj.p();
obj.m();
// We can refer to ... obj.x ... obj.y ...
// ... obj.u ... obj.v1 ... obj.v2 ...
```



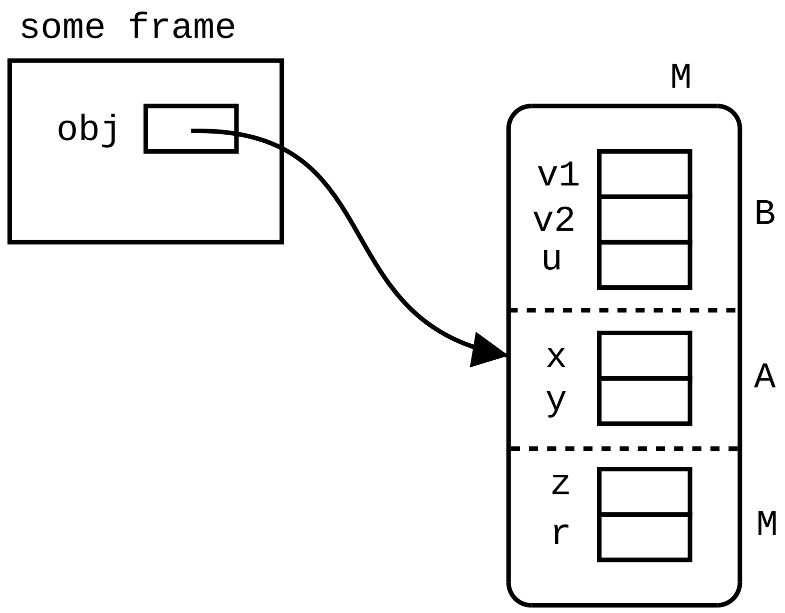
Inheritance

- A method in a subclass can access the attributes and methods of its super class.

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... v1 ... v2 ... u ... m() ... }
}
class A extends B {
  E x;
  C y;
  void p()
  {
    ... x ... y ... p() ... v1 ...
    ... v2 ... u ... m() ...
  }
  void s() { ... }
}
```

Inheritance

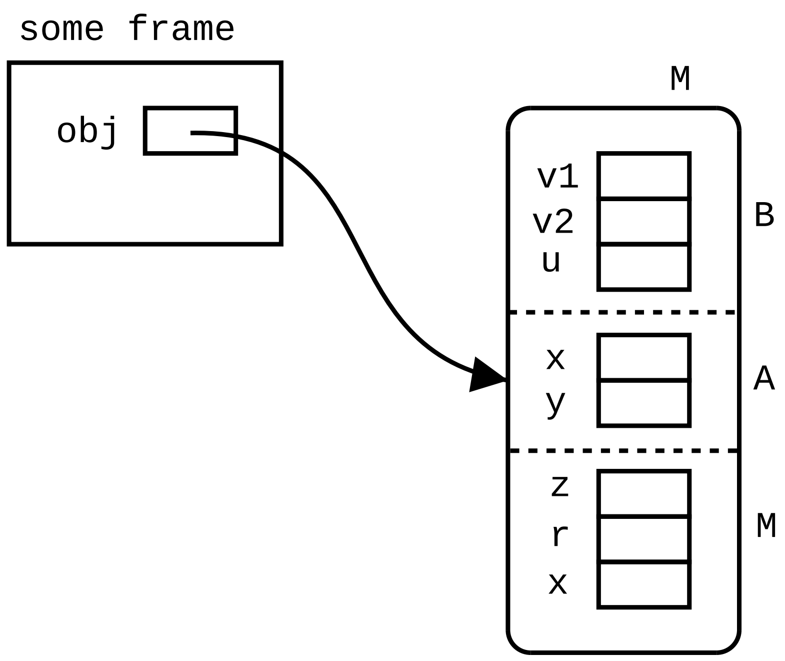
```
class M extends A {
  E z;
  D r;
  void q() { ... }
}
// Somewhere else
M obj2 = new M();
```



Shadowing variables

- An attribute or instance variable can be redefined in a subclass. In this case we say that the variable in the subclass *shadows* the variable in the parent class.

```
class M extends A {
  E z;
  D r, x;
  void q() { ... }
}
```



Accessing variables from the super class

- The `super` reference is used to access an attribute or method in a parent class.

```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
}
```

Overriding methods

- A method can be redefined in a subclass. This is called *overriding* the method.

```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
  void p()
  {
    ...
  }
}
```

Inheritance

- A method in a subclass can access the attributes and methods of its super class.

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... v1 ... v2 ... u ... m() ... }
}
class A extends B {
  E x;
  C y;
  void p()
  {
    ... x ... y ... p() ... v1 ...
    ... v2 ... u ... m() ...
  }
  void s() { ... }
}
```

Accessing a method or attribute

- When we try to access a method or attribute of an object, it is looked up by the Java runtime system in the class of the object first. If it is not found there, it is looked up in the parent class. If it is not found there, it is looked up in the grand-parent, etc...

```
M obj3 = M();
obj3.q();  // From class M
obj3.m();  // From class B
obj3.p();  // From class M
obj3.s();  // From class A
```

- Attributes and methods declared as `private` cannot be accessed directly by the subclasses, even though they are present in the object. They can be accessed only indirectly by public accessor methods in the class that declared them as private.

Accessing a method or attribute

```
class A extends B {
  private E x, y;
  void p() { }
  void s() { }
  public E get_x() { return x; }
}
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ...
    // instead of super.x ...
    ... get_x() ... or ... super.get_x() ...
  }
}
```

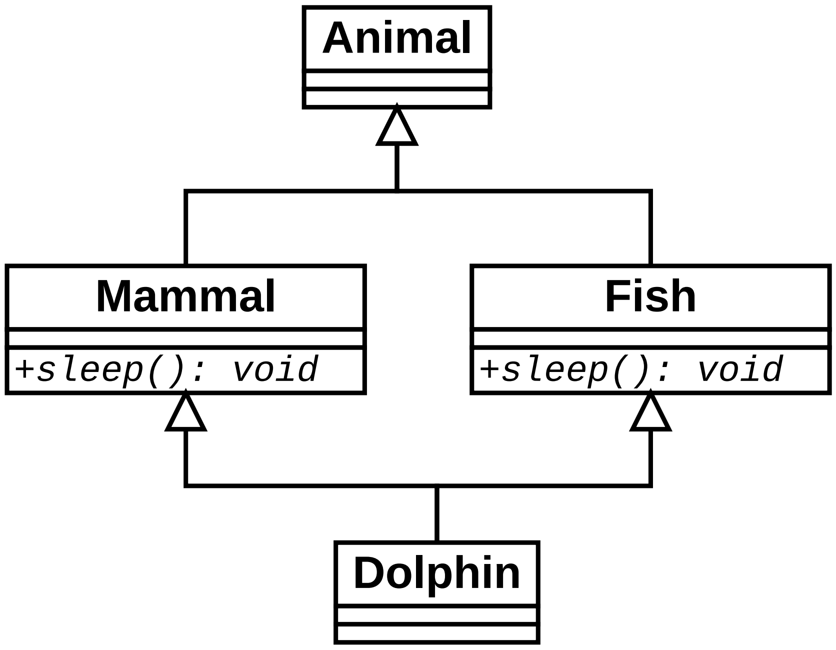
Accessing a method or attribute

- An attribute or method declared as `protected` can be accessed by any subclass, even if it is in a different package.
- An attribute or method declared as `final`, is not inherited at all, i.e. it forbids overriding.
- A class declared as `final`, cannot have subclasses.

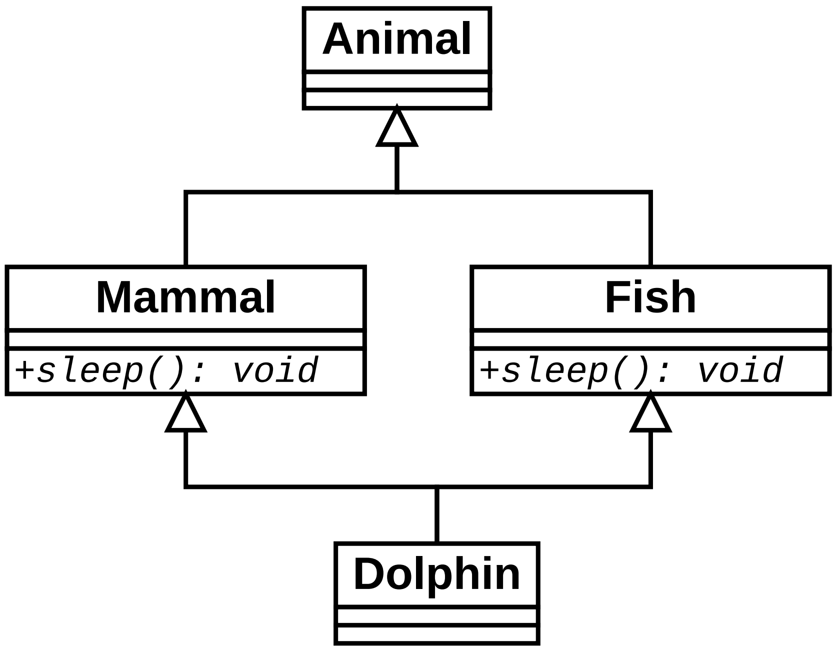
Multiple inheritance

- Multiple inheritance: a class with more than one superclass

Multiple inheritance



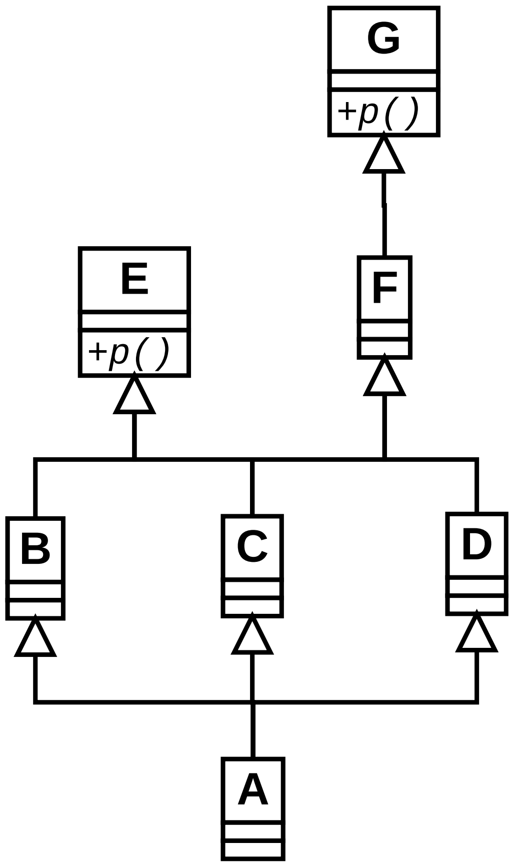
Multiple inheritance



```
class A extends B, C { ... } // Error
```

- Java does not support multiple inheritance

Multiple inheritance



Polymorphism

- Polymorphism means "many forms."
- Polymorphism is the characteristic of being able to assign a different meaning or usage to something in different contexts
- A polymorphic method is a method which can accept more than one type of argument
- Kinds of polymorphism:
  - Overloading (Ad-hoc polymorphism): redefining a method in the same class, but with different signature (multiple methods with the same name.) Different code is required to handle each type of input parameter.
  - Parametric polymorphism: a method is defined once, but when invoked, it can receive as arguments objects from any subclass of its parameters. The same code can handle different types of input parameters.

Polymorphism

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...’’);
  }
}
class Human extends Creature {
  void move()
  {
    System.out.println("Walking...");
  }
}
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
}
```

Ad-hoc Polymorphism (Overloading)

```
class Zoo {
  void animate(Human h)
  {
    h.move();
  }
  void animate(Martian m)
  {
    m.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick);  // Polymorphic call
  }
}
```

Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick);  // Polymorphic call
  }
}
```

Accessing super

```
class Human extends Creature {
  void move()
  {
    super.move();
    System.out.println(“Walking...”);
  }
}
class Martian extends Creature {
  void move()
  {
    super.move()
    System.out.println(“Crawling...”);
  }
}
```

Casting and instanceof

- Casting is like putting a special lens on an object
- A casting expression is of the form

```
(type) expr
```

where `type` is any type (primitive or user-defined) and `expr` is an expression which evaluates to an object reference whose type is compatible with `type`.

- Not all casts are possible

```
(int) ‘‘Hello’’
(Engine) yannick
```

Casting

- If a variable is a reference of type A, it can be assigned any object whose type is a subclass of B.

```
Human greg = new Human();
Creature c = greg;
```

- But a reference of type B cannot be assigned directly reference of type A, if B is a subclass of A (because A has less attributes than required by B):

```
Creature d = new Creature();
Martian m = d;
```

- ...however, if we know that a reference x of type A points to an object of type B (and B is a subclass of A,) then we can force to see x as being of type B by using a casting expression:

```
Creature e = new Martian();
Martian f = (Martian)e;
```

Checking the type of a reference

- To find out whether a reference `r` is an instance of a particular class `C` we use the boolean expression:

```
r instanceof C
```

- This is normally used whenever we do casting:

```
class Human extends Creature {
  void move()
  {
    System.out.println(‘‘Walking...’’);
  }
  void jump()
  {
      System.out.println("Up and down");
  }
}
```

McGill

Checking the type of a reference

```
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
  void hop()
  {
      System.out.println("Down and to the left");
  }
}
class Zoo {
  void move(Creature c)
  {
    if (c instanceof Human)
      ((Human)c).jump();
    else if (c instanceof Martian)
      ((Martian)c).hop();
    c.move();
  }
}
```

The end